Optimize4: An optimization package for *morpho*

June 10, 2024

Chapter 1

Overview

The `optimize4` package faciliates the solution of optimization problems, with a particular focus on the shape optimization problems for which *morpho* was designed. The design is intended to be flexible, enabling customization of the choice of algorithm and easy incorporation of new algorithms by the developer or user. To use the package, simply import it into your *morpho* program as usual:

```
import optimize4
```

This imports several subsidiary modules, which provide three main kinds of class that work together (Fig. 1):

OptimizationProblem classes are used to describe the problem to be solved. Functionals may be added to the problem either as energies or constraints with set target values.

OptimizationAdapter classes provide a uniform interface for optimization targets, enabling the setting and getting parameters as well as calculating the value of the objective function, constraints and gradients. Adapters are provided, for example, to take an *OptimizationProblem* and a target object, such as a Mesh or a Field, and compute the value of the objective function and gradients with respect to the target. Adapters can also be used to transform one type of problem to another, e.g. a constrained problem to an unconstrained problem, facilitating the use of different optimization algorithms.

OptimizationController classes objects implement an optimization algorithm or a useful subcomponent. Controllers work by calling appropriate methods on associated OptimizationAdapters to obtain value,

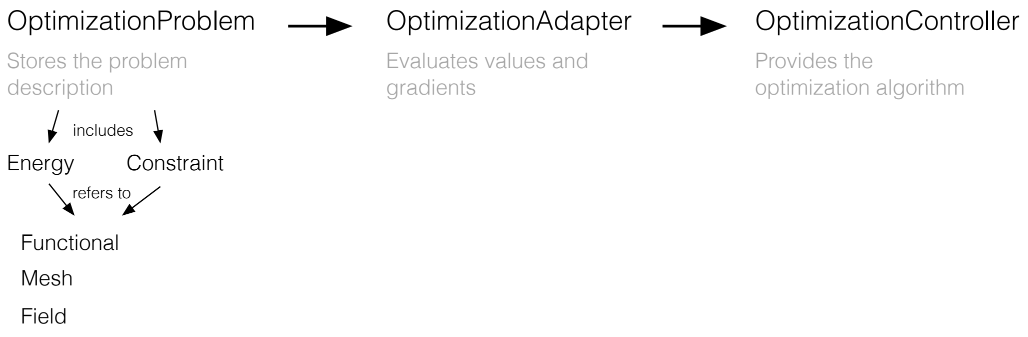

Figure 1.0.1: **Classes in the optimize4 package** and how they interact.

gradient and even Hessians in some cases, and direct how parameters are to be adjusted as the algorithm proceeds.

Chapter 2

Using optimize

To establish notation, the goal of the `optimize4` package is to solve the following standard problem,

$$\begin{aligned} &\min_{\mathbf{x}} f(\mathbf{x}) \\ &s.t.\ \mathbf{c}(\mathbf{x}) = 0 \\ &\quad\ \ \mathbf{d}(\mathbf{x}) \geq 0 \end{aligned} \tag{2.0.1}$$

where $f(\mathbf{x})$ is the objective function and $\mathbf{x}$ are its N parameters. There are a total of M constraints, expressed as $\mathbf{c}(\mathbf{x})$ a vector of $M_=$ equality constrained functions, and $\mathbf{d}$ a vector of $M_{\neq} = M - M_=$ inequality constrained functions. We adopt the sign convention that the inequality constraint functions $\mathbf{d}$ are positive in the feasible region.

We shall use the subscript notation $\mathbf{x}_k$ to refer to the value of a quantity at a particular iteration k. If $\mathbf{c}$ and $\mathbf{d}$ are empty vectors the problem is said to be *unconstrained*. If constraints are present, the set $\mathbf{x} \in \Omega$ compatible with the constraints is called the *feasible* set.

We shall also use consistent notation for the gradient of the objective function,

$$\mathbf{g} = \frac{\partial f}{\partial x_i}$$

and its hessian,

$$\mathbf{H} = H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

as well as the $N \times N_c$ matrix of the gradients of constraints,

$$\mathbf{C} = C_{ij} = \frac{\partial c_j}{\partial x_i},\ \mathbf{D} = D_{ij} = \frac{\partial d_j}{\partial x_i}$$

[More to go here]

Chapter 3

OptimizationProblem

An OptimizationProblem is a container object that describes an optimization problem using Functional objects, which can be used as part of the objective function (these are referred to as “energies”), or as constraints. Earlier versions of *morpho* provided the same user interface, which has been adopted and integrated into the `optimize4` package. Creation of an OptimizationProblem is described in the main *morpho* manual; we provide a simple example below appropriate for minimizing the length of a closed loop at constant enclosed area:

```
var problem = OptimizationProblem(mesh)

problem.addenergy(Length())
problem.addconstraint(EnclosedArea())
```

Chapter 4

OptimizationAdapters

OptimizationAdapter and its subclasses exist to provide a uniform interface for OptimizationControllers to call. This design enables optimization algorithms to be separated from the task of evaluating quantities like gradients and widens the range of things that could be optimized—all that's needed to optimize an arbitrary object that depends on parameters is to create an appropriate OptimizationAdapter.

Further, adapter objects can be chained together to create useful effects. An adapter such as PenaltyAdapter converts a constrained problem to an unconstrained problem, for example, facilitating use of a different set of OptimizationControllers. Appropriate adapters can also be used within a broader algorithm to solve a subproblem of interest, for example performing a linesearch on an augmented objective function rather than the original one.

4.1 OptimizationAdapter interface

An adapter **must** implement the following methods,

set(x) Sets the current value of the parameters to x, which should be supplied as a column vector.

get() Returns the current value of the parameters as a column vector.

value() Returns the value of the objective function.

gradient() Returns the gradient of the objective function at the current parameters as a column vector.

hessian() Returns the hessian of the objective function at the current parameters as a column vector, or `nil` if a hessian is not available.

countConstraints() Returns the total number of constraints present $M = M_{=} + M_{\neq}$.

countEqualityConstraints() Returns the number of equality constraints $M_{=}$.

countInequalityConstraints() Returns the number of equality constraints $M_{\neq}$.

constraintValue() Returns a List containing the value(s) of any constraints.

constraintGradient() Returns a List containing the gradient(s) of any constraints as column vectors.

constraintHessian() Returns a List containing the hessian(s) of any constraints.

4.2 Available adapters

4.2 DelegateAdapter

A DelegateAdapter is initialized with a given adapter. The DelegateAdapter implements the Optimization-Adapter interface, but simply redirects all of method calls to the enclosed adapter.

4.2 ProxyAdapter

Implements the *proxy* software design pattern[1] for the adapter protocol. Calls to `value`, `gradient`, etc. are returned from a cache if they have already been calculated, or are calculated as necessary. Every time `set` is called with new parameters, the cache is reset. This adapter prevents multiple evaluation of potentially expensive quantities like the gradient or the hessian by OptimizationControllers. Using a ProxyAdapter helps simplify the writing a controller: there's no need to temporarily store these quantities across methods, for example.

A ProxyAdapter also keeps track of how many times the objective function value, gradient etc. are actually calculated and can display this information using the `report` method. This information can also be obtained as a list in the order $(N_f, N_{\mathbf{g}}, N_{\mathbf{H}}, N_{\mathbf{c}}, N_{\nabla\mathbf{c}}, N_{\Delta\mathbf{c}})$ from the `countEvals` method.

4.2 FunctionAdapter

A FunctionAdapter provides an interface to minimize a callable *morpho* object, i.e. a function, invocation or closure, with respect to its positional parameters. To miminize a simple quadratic function $(x-1/2)^2+(y-1)^2+\frac{1}{4}xy$, for example,

```
fn func(x, y) {
    return (x-0.5)^2 + (y-1)^2 + 0.25*x*y
}

var adapt = FunctionAdapter(func, start=Matrix(2)) // Start from (0,0)
```

You can specify the starting point through the `start` optional parameter in the constructor, and provide functions that return the gradient and/or hessian via `gradient` and `hessian`. If these are not provided, FunctionAdapter will compute approximations using finite differences.

If you want to optimize a callable object with respect to its optional parameters, you can wrap it in another function or closure:

```
fn wrapper(x, y) {
    return myfunc(a=x, b=y)
}

var adapt = FunctionAdapter(wrapper, start=Matrix(2)) // Start from (0,0)
```

OptimizationControllers

In this chapter we review optimization algorithms available in the package, which are implemented as OptimizationControllers, with enough context to understand their relative utility. For a deeper understanding of these algorithms, the reader should consult standard textbooks on optimization theory[2].

[1]Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 207ff

[2]Recommended texts include Nocedal and Wright, "Numerical Optimization" (Springer); Boyd and Vandenberghe "Convex Optimization" (Cambridge University Press).

New OptimizationControllers can easily be defined, including ones created in extensions linking to external optimization libraries.

4.2 PenaltyAdapter

A PenaltyAdapter can be used to convert the general constrained problem (2.0.1) into an unconstrained problem. It must be constructed with another adapter as the target, and reports the value and derivatives of the new objective function,

$$\mathcal{L} = f(\mathbf{x}) + \mu \left|\mathbf{c}\right|^2 + \mu \left|\mathbf{d}^-\right|^2 , \tag{4.2.1}$$

where $\mathbf{d}^-$ is defined for each component $d_i^- = argmin(0, d_i)$, i.e. nonzero for constraints *outside* the feasible region. The parameter μ is called a *penalty parameter*.

Using this adapter facilitates penalty methods, that solve $\min_{\mathbf{x}} \mathcal{L}(\mu_m)$ for monotonically increasing penalty parameters μ_m. As $\mu \to \infty$, the solution of the new unconstrained problem should converge on the constrained problem.

PenaltyAdapter relies on the attached adapter to provide value, gradient and hessian information if requested by the controller; note especially that the hessian of the constraint functions, and not just their gradient, is required to compute the hessian of $\mathcal{L}$ in 4.2.1.

Chapter 5

OptimizationControllers

In this chapter we review optimization algorithms available in the package, which are implemented as OptimizationController objects, with enough context to understand their relative utility. For a deeper understanding of these algorithms, the reader should consult standard textbooks on optimization theory[1].

New classes that implement OptimizationController protocols can easily be defined, including ones created in extensions linking to external optimization libraries.

5.1 Reporting

OptimizationController objects support controllable levels of output. To set, initialize the OptimizationController with the optional parameter `verbosity` with one of the following options, supplied as a String

silent Suppresses all output except errors.

quiet Suppresses all output except errors and warnings.

normal Report progress of the controller, but suppress additional output

verbose Display all available information

5.2 Interface

An OptimizationController is a base class for iterative optimization algorithms. The user performs optimization by calling the `optimize` method with a maximum number of iterations.

The generic optimization sequence is shown in Algorithm 5.1. Particular algorithms are defined by subclassing OptimizationController and implementing the below methods, all of which do nothing in the base class unless otherwise indicated.

start() Called once by `optimize` to performs any initialization at the beginning of optimization.

begin() Calculate any necessary quantities at the beginning of each iteration.

step() Perform the optimization step

next() Calculate updated information at the end of the iteration.

[1]Recommended texts include Nocedal and Wright, "Numerical Optimization" (Springer); Boyd and Vandenberghe "Convex Optimization" (Cambridge University Press).

Algorithm 5.1 Generic optimization algorithm

```
1: start()
2: record()
3: for $i \leftarrow 1$ to $N_{iter}$ do
4:     if hasConverged() then
5:         break
6:     end if
7:     iterate()
8:     report(i)
9:     record()
10: end for
```

iterate() Calls `begin`, `step` and `next` in turn.

record() Records information about the iteration. By default, the value of the objective functional is stored after each iteration step.

The OptimizationController base class provides two further methods that form part of the generic optimization algorithm:

hasConverged() Performs a convergence check as described in subsection 5.3 below.

report(i) Reports information to the user, including the value of the optimization algorithm

5.3 Convergence criteria

The base OptimizationController class provides two basic convergence criteria, that are controlled by the properties `gradtol` and `etol`.

1. The first criterion is to examine the norm of the gradient of the objective function,

$$|g_k| < \texttt{gradtol}$$

   where `gradtol` is 1×10^{-6} by default.

2. The second is to monitor the change in the value of the objective function in successive iterations of the algorithm, f_k and f_{k+1} and stop if,

$$\begin{cases} |f_{k+1} - f_k| < \texttt{etol}, & |f_{k+1}| < \texttt{etol} \\ |f_{k+1} - f_k| < \texttt{etol}\,|f_{k+1}|\,, & \text{otherwise} \end{cases}$$

   i.e. ensuring a relative tolerance of `etol` unless f itself is nearly zero, in which case `etol` is an absolute tolerance. The value of `etol` is 1×10^{-8} by default.

The method `hasConverged` returns **`true`** if either of these convergence checks are met.

5.4 Gradient descent methods

Gradient descent methods select a search direction $\mathbf{d}_k$ and update it according to the rule,

$$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \gamma_k \mathbf{d}_k$$

where γ_k is the *stepsize*.

5.4 GradientDescentController

The simplest choice, called *steepest descent*, is implemented by the GradientDescentController class. It simply uses the negative of the gradient of f as the search direction, i.e. $\mathbf{d}_k = -\mathbf{g}_k$ and $\gamma_k = \gamma$ constant for all iterations. This method is cheap per iteration, requiring a single evaluation of the gradient, but typically converges very slowly indeed and may oscillate around the minimum in some cases. The stepsize γ can be set in the initializer, or by changing the property `stepsize`; the default value is 0.1.

```
var control = GradientDescentController(adapter, stepsize=0.1)
```

While GradientDescentController is rarely useful directly, it is sometimes helpful in very constrained problems (where linesearches may offer little benefit) or to establish that a new OptimizationAdapter is indeed working correctly.

5.4 LineSearchController

An improvement in performance can be obtained by considering the one parameter subproblem,

$$\min_{\gamma_k} f(\mathbf{x}_k + \gamma_k \mathbf{d}_k) \tag{5.4.1}$$

A LineSearchController aims to find a stepsize γ_k at each iteration that approximately solves this subproblem. The search direction $\mathbf{d}_k$ is by default chosen to be the steepest descent direction $\mathbf{d}_k = -\mathbf{g}_k$, but could be chosen by some other method either by subclassing LineSearchController and replacing the `searchdirection` method, or by performing linesearches one at a time by setting the `direction` property and invoking the `step` method.

Typically, it's not necessary to solve (5.4.1) very precisely. LineSearchController therefore implements a *backtracking line search*, starting with $\gamma_k = 1$ and then successively reducing it by a factor $\beta \in (0, 1]$ at a time (so that $\gamma_k = \beta^n$) until the *Armijo condition*,

$$f(\mathbf{x}_k + \gamma_k \mathbf{d}_k) < f(\mathbf{x}_k) + \alpha \mathbf{g}_k \cdot \mathbf{d}_k,$$

is met, where $\alpha \in (0, 1)$ is a parameter. Success indicates that the function has decreased at least proportionately to what might be predicted from the gradient, and hence this condition helps to prevent "overshooting" the minimum. Default parameters are $\alpha = 0.2$ and $\beta = 0.5$; these can be adjusted by setting the `alpha` and `beta` in the LineSearchController's initializer or by setting the properties directly.

There is a limit placed on the number of backtracking steps in a property called `maxsteps`; if this is exceeded a warning `OptLnSrchStpsz` is generated. The LineSearchController also checks to ensure that $\mathbf{g}_k \cdot \mathbf{d}_k \leq 0$, i.e. that the search direction is actually downward. If this is *not* the case, a warning `OptLnSrchDrn` is raised and it is likely that there is an error somewhere, either in the gradient calculation (if at some stage you provided one) or in the algorithm that calculated the search direction.

5.4 ConjugateGradientController

The conjugate gradient algorithm performs linesearches, but computes the search direction using information from the prior iteration. The resulting algorithm improves convergence on objective functions that have long, narrow valleys and uses gradient information only. As for gradient descent the search direction is initially,

$$\mathbf{d}_0 = -\mathbf{g}_0,$$

and then in successive iterations the direction is computed,

$$\mathbf{d}_{k+1} = -\mathbf{g}_{k+1} + \beta_k \mathbf{d}_k.$$

The scaling β_k may be computed in several different ways, leading to different flavors of the algorithm. For example, the Fletcher-Reeves formula is,

$$\beta_k = \frac{\mathbf{g}_{k+1} \cdot \mathbf{g}_{k+1}}{\mathbf{g}_k \cdot \mathbf{g}_k}.$$

5.5 Newton and Quasi-Newton methods

Newton's method utilizes the observation that around a given point $\mathbf{x}_k$, a sufficiently smooth function can be approximated by a Taylor expansion,

$$f(\mathbf{x}) \approx f(\mathbf{x}_k) + \mathbf{g} \cdot (\mathbf{x} - \mathbf{x}_k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T \cdot \mathbf{H} \cdot (\mathbf{x} - \mathbf{x}_k) + ... \tag{5.5.1}$$

Taking the gradient of (5.5.1) yields,

$$\nabla f(\mathbf{x}) = \mathbf{g} \cdot (\mathbf{x} - \mathbf{x}_k) + \mathbf{H} \cdot (\mathbf{x} - \mathbf{x}_k) + ...$$

At the minimum of f, its gradient should be zero, i.e. that,

$$0 = \mathbf{g} \cdot (\mathbf{x} - \mathbf{x}_k) + \mathbf{H} \cdot (\mathbf{x} - \mathbf{x}_k) + ...$$

and hence, neglecting higher order terms, we can obtain an update $(\mathbf{x} - \mathbf{x}_k)$ by solving the linear system,

$$\mathbf{H}_k \cdot (\mathbf{x} - \mathbf{x}_k) = -\mathbf{g_k}, \tag{5.5.2}$$

which is called a Newton step. If $f(\mathbf{x})$ is a quadratic function, then Eq. 5.5.1 is no longer an approximation and the correct solution can be obtained in one iteration. In practice, the solution of Eq. 5.5.2 is used as a search direction $\mathbf{d}_k$ for a line search.

5.5 NewtonController

The NewtonController class implements Newton updates: in each iteration a search direction is identified by solving Eq. 5.5.2 and then a linesearch is performed. NewtonController requires an OptimizationAdapter that can calculate a hessian and raises the error `OptNoHess` it it detects this is not the case.

5.5 BFGSController

The BFGS algorithm aims to compute an improved estimate of the Hessian at each iteration using only the change in parameter values $\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and the change in gradient $\mathbf{y}_k = \mathbf{g}_{k+1} - \mathbf{g}_k$. The estimate is initially set to the identity matrix,

$$\mathcal{H}_0 = \mathbf{I},$$

and then updated after each iteration using the formula,

$$\mathcal{H}_{k+1} = \mathcal{H}_k + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \mathbf{s}_k} - \frac{\mathcal{H}_k \mathbf{s}_k \mathbf{s}_k^T \mathcal{H}_k^T}{\mathbf{s}_k^T \mathcal{H}_k \mathbf{s}_k}.$$

At each iteration, the current BFGS estimate of the Hessian is used to solve the linear system,

$$\mathcal{H}_k \mathbf{d}_k = -\mathbf{g}_k, \tag{5.5.3}$$

to obtain a search direction $\mathbf{d}_k$ that is then used as the basis of a linesearch.

5.5 InvBFGSController

This controller implements a variant of the BFGS algorithm that avoids the need to explicitly solve (5.5.3); instead successive estimates of the inverse Hessian, $\mathcal{H}_k^{-1}$, are built from an initial identity matrix.

5.5 LBFGSController

The LBFGS algorithm avoids the need to store an estimated inverse Hessian explicitly, instead computing its action from a stored set of $\mathbf{s}_k$ and $\mathbf{y}_k$.

5.6 Constrained optimization

5.6 PenaltyController

Implements a penalty method. Initialize with a constrained problem, and (optionally) an initial value of the penalty parameter `mu0` and a multiplier `mumul`.

```
var control = PenaltyController(adapter, mu0=1, mumul=10)
```

Internally, PenaltyController uses a PenaltyAdapter to convert the constrained problem into an unconstrained form. Successive iterations increase the penalty parameter until the constraint tolerance `ctol` is met. To perform optimization for each value of μ, PenaltyController uses a nested controller which is by default an LBFGSController, but you can choose to use an alternative by providing the PenaltyController constructor with the appropriate class as an optional argument `controller`.