# Optimize3: An optimization package for *morpho*

February 2, 2024

# Chapter 1

# Overview

The `optimize3` 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 optimize3
```

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,

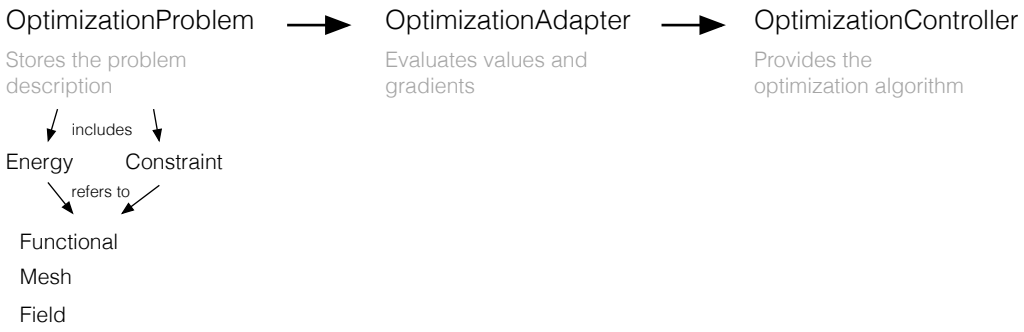


Figure 1.0.1: **Classes in the optimize3 package** and how they interact.

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

# Chapter 2

# Using optimize3

To establish notation, the goal of the `optimize3` package is to solve the following standard problem,

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

where $f(\mathbf{x})$ is the objective function, $\mathbf{x}$ are its parameters, $\mathbf{c}(\mathbf{x})$ is a vector of equality constrained functions, and $\mathbf{d}$ is a vector of inequality constrained functions.

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,

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

*[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 `optimize3` 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 connected 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 could 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 Available adapters

### 4.1 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)
```

## 4.1 MeshAdapter

A MeshAdapter evaluates the value and gradient of a given OptimizationProblem with respect to a Mesh's degrees of freedom; both must be supplied in the constructor. The current configuration of the mesh is used as the starting point.

```
var adapt = MeshAdapter(problem, mesh)
```

## 4.1 FieldAdapter

A FieldAdapter evaluates the value and gradient of a given OptimizationProblem with respect to a Field's degrees of freedom; both must be supplied in the constructor. The current configuration of the field is used as the starting point. Only functionals in the OptimizationProblem that actually depend on the supplied field are included in the objective function.

```
var adapt = FieldAdapter(problem, field)
```

## 4.1 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 may help 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 `countevaluations` method.

## 4.1 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,

$$\mathcal{L} = f(\mathbf{x}) + \mu \left|\mathbf{c}\right|^2 + \mu \left|\mathbf{d}^-\right|^2, \tag{4.1.1}$$

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 $\mu$ 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 $\mu_m$. 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 4.1.1.

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

### 4.1 LagrangeMultiplierAdapter

A LagrangeMultiplierAdapter converts the general constrained problem (2.0.1) into the new unconstrained objective function,

$$\mathcal{L} = f(\mathbf{x}) + \lambda^T \cdot \mathbf{c} + \lambda'^T \cdot \mathbf{d}^-$$

where $\lambda$ and $\lambda'$ are vectors of Lagrange multipliers, one for each constraint present in the original problem.

### 4.1 AugmentedLagrangianAdapter

An AugmentedLagrangianAdapter converts the general constrained problem (2.0.1) into the new unconstrained objective function,

$$\mathcal{L} = f(\mathbf{x}) + \mu \left|\mathbf{c}\right|^2 + \mu \left|\mathbf{d}^-\right|^2 + \lambda^T \cdot \mathbf{c} + \lambda'^T \cdot \mathbf{d}^-$$

where $\lambda$ and $\lambda'$ are vectors of Lagrange multipliers, one for each constraint present in the original problem and $\mu$ is a penalty parameter. The resulting formulation combines elements of Lagrange multiplier and penalty methods; algorithms that use these adapters update both $\mu$ and $\lambda$ as they converge on a solution.

## 4.2 OptimizationAdapter interface

An adapter **must** implement the following methods, except where marked optional.

### 4.2 set(x)

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

### 4.2 get()

Returns the current value of the parameters as a column vector.

### 4.2 value()

Returns the value of the objective function.

### 4.2 gradient()

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

### 4.2 hessian()

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

### 4.2 countconstraints()

Returns the number of constraints present. In future, this should be a Tuple with the first element being the number of equality constraints and the second being the number of inequality constraints.

### 4.2 constraintvalue()

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

## 4.2 constraintgradient()

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

# Chapter 5

# 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.

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

## 5.1 Convergence criteria

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

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

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

   where `gradtol` is $1 \times 10^{-6}$ 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,

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

   i.e. ensuring a relative tolerance of `etol` unless $f$ itself is nearly zero, in which case `etol` is an absolute tolerance.

If, during optimization either of these convergence criteria are true, then the optimization terminates.

## 5.2 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 $\gamma_k$ is the *stepsize*.

## 5.2 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 $\gamma$ 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 offer little benefit) or to establish that a new OptimizationAdapter is indeed working correctly.

## 5.2 LineSearchController

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

$$\min_{\alpha_k} f(\mathbf{x}_k + \gamma_k \mathbf{d}_k) \tag{5.2.1}$$

A LineSearchController aims to find a stepsize $\gamma_k$ 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 (5.2.1) 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.

## 5.2 ExactLineSearchController

While it is rarely valuable to solve the line search subproblem (5.2.1) to high precision, an ExactLineSearchController will be provided to do so if necessary. This controller performs a 1D minimization using Brent's algorithm.

## 5.3 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,

$$f(\mathbf{x}) \approx f(\mathbf{x}_k) + \mathbf{g} \cdot (\mathbf{x} - \mathbf{x}_k) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T \cdot \mathbf{H} \cdot (\mathbf{x} - \mathbf{x}_k) + ... \tag{5.3.1}$$

Taking the gradient of (5.3.1) 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,

$$\mathbf{H}_k \cdot (\mathbf{x} - \mathbf{x}_k) = -\mathbf{g_k}, \tag{5.3.2}$$

which is called a Newton step. If $f(\mathbf{x})$ is a quadratic function, then Eq. 5.3.1 is no longer an approximation and the correct solution can be obtained in one iteration. In practice, the solution of Eq. 5.3.2 is used as a search direction $\mathbf{d}_k$ for a line search.

### 5.3 NewtonController

The NewtonController class implements Newton updates: in each iteration a search direction is identified from Eq. 5.3.2 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.

### 5.3 ConjugateGradientController

### 5.3 BFGSController

The BFGS algorithm aims to compute an estimate of the Hessian, or it's inverse, using

### 5.3 LBFGSController

## 5.4 Constrained optimization