Flexible numerical optimization with ensmallen

Ryan R. Curtin\textsuperscript{1,\ast}, Marcus Edel\textsuperscript{2,3,\ast}, Rahul Ganesh Prabhu\textsuperscript{4}, Suryoday Basak\textsuperscript{5}, Zhihao Lou\textsuperscript{6}, Conrad Sanderson\textsuperscript{7,8}

\textsuperscript{1} RelationalAI, Atlanta, USA
\textsuperscript{2} Free University of Berlin, Germany
\textsuperscript{3} Collabora, Montreal, Canada
\textsuperscript{4} Birla Institute of Technology and Science, Pilani, Pilani Campus, India
\textsuperscript{5} Department of Computer Science and Engineering, The University of Texas at Arlington, USA
\textsuperscript{6} Epsilon, Chicago, USA
\textsuperscript{7} Data61 / CSIRO, Australia
\textsuperscript{8} Griffith University, Australia
\textsuperscript{\ast} Primary maintainers

Abstract

This report provides an introduction to the ensmallen numerical optimization library, as well as a deep dive into the technical details of how it works. The library provides a fast and flexible C++ framework for mathematical optimization of arbitrary user-supplied functions. A large set of pre-built optimizers is provided, including many variants of Stochastic Gradient Descent and Quasi-Newton optimizers. Several types of objective functions are supported, including differentiable, separable, constrained, and categorical objective functions. Implementation of a new optimizer requires only one method, while a new objective function requires typically only one or two C++ methods. Through internal use of C++ template metaprogramming, ensmallen provides support for arbitrary user-supplied callbacks and automatic inference of unsupplied methods without any runtime overhead. Empirical comparisons show that ensmallen outperforms other optimization frameworks (such as Julia and SciPy), sometimes by large margins. The library is available at https://ensmallen.org and is distributed under the permissive BSD license.

1 Introduction

The problem of mathematical optimization is fundamental in the computational sciences [54]. In short, this problem is expressed as

\[
\arg\min_x f(x)
\]

(1)

where \( f(x) \) is a given objective function and \( x \) is typically a vector or matrix. The ubiquity of this problem gives rise to the proliferation of mathematical optimization toolkits, such as SciPy [81], opt++ [48], OR-Tools [58], CVXOPT [76], NLopt [28], Ceres [3], and RBFOpt [13]. Furthermore, in the field of machine learning, many deep learning frameworks have integrated optimization components. Examples include Theano [4], TensorFlow [1], PyTorch [56], and Caffe [26].

Mathematical optimization is generally quite computationally intensive. For instance, the training of deep neural networks is dominated by the optimization of the model parameters on the data [36, 39]. Similarly, other popular machine learning algorithms such as logistic regression are also expressed as and dominated by an optimization process [85, 46]. Computational bottlenecks occur even in fields as wide-ranging as rocket landing guidance systems [19], motivating the development and implementation of specialized solvers.

The necessity for both efficient and specializable function optimization motivated us to implement the ensmallen library, originally as part of the mlpack machine learning library [14]. The ensmallen library provides a large set of pre-built optimizers for optimizing user-defined objective functions in C++; at the time of writing, 46 optimizers are available. The external interface to the optimizers is intuitive and matches the ease of use of popular optimization toolkits mentioned above.

However, unlike many of the existing optimization toolkits, ensmallen explicitly supports numerous function types: arbitrary, differentiable, separable, categorical, constrained, and semidefinite. Furthermore, custom behavior during optimization can be easily specified via callbacks. Lastly, the underlying framework facilitates the implementation of new optimization techniques, which can be contributed upstream and incorporated into the library. Table 1 contrasts the functionality supported by ensmallen and other optimization toolkits.
This report is a revised and expanded version of our initial overview of ensmallen [10]. It also provides a deep dive into the internals of how the library works, which can be a useful resource for anyone looking to contribute to the library or get involved with its development.

We continue the report as follows. We overview the available functionality and show example usage in Section 2. Advanced usage such as callbacks is discussed in Section 3. We demonstrate the empirical efficiency of ensmallen in Section 4. The salient points are summarized in Section 5.

## 2 Overview

The task of optimizing an objective function with ensmallen is straightforward. The class of objective function (e.g., arbitrary, constrained, differentiable, etc.) defines the implementation requirements. Each objective function type has a minimal set of methods that must be implemented. Typically this is only between one and four methods. As an example, to optimize an objective function $f(x)$ that is differentiable, implementations of $f(x)$ and $f'(x)$ are required. One of the optimizers for differentiable functions, such as L-BFGS [41], can then be immediately employed.

Whenever possible, ensmallen will automatically infer methods that are not provided. For instance, given a separable objective function $f(x) = \sum_i f_i(x)$ where an implementation of $f_i(x)$ is provided (as well as the number of such separable objectives), an implementation of $f(x)$ can be automatically inferred. This is done at compile-time, and so there is no additional runtime overhead compared to a manual implementation. C++ template metaprogramming techniques [2, 5, 80] are internally used to produce efficient code during compilation.

Not every type of objective function can be used with every type of optimizer. For instance, since L-BFGS is a differentiable optimizer, it cannot be used with a non-differentiable object function type (e.g. an arbitrary function). When an optimizer is used with a user-provided objective function, an internal mechanism automatically checks the requirements, resulting in user-friendly error messages if any required methods are not detected.

<table>
<thead>
<tr>
<th>Library</th>
<th>unified framework</th>
<th>constraints</th>
<th>separable functions / batches</th>
<th>arbitrary functions</th>
<th>arbitrary optimizers</th>
<th>sparse gradients</th>
<th>categorical</th>
<th>arbitrary types</th>
<th>callbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ensmallen</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Shogun [70]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Vowpal Wabbit [38]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>TensorFlow [1]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>PyTorch [56]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Caffe [26]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Keras [12]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>scikit-learn [57]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>SciPy [29]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>MATLAB [47]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Julia (Optim.jl) [49]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Table 1: Feature comparison: ● = provides feature, ○ = partially provides feature, - = does not provide feature. unified framework indicates if there is a form of generic/unified optimization framework; constraints and separable functions / batches indicate support for constrained functions and separable functions; arbitrary functions means arbitrary objective functions are easily implemented; arbitrary optimizers means arbitrary optimizers are easily implemented; sparse gradient indicates that the framework can natively take advantage of sparse gradients; categorical refers to if support for categorical features exists; arbitrary types mean that arbitrary types can be used for the parameters $x$; callbacks indicates that user-implementable callback support is available.
2.1 Types of Objective Functions

In most cases, the objective function \( f(x) \) has inherent attributes; for example, as mentioned in the previous paragraphs, \( f(x) \) might be differentiable. The internal framework in \( \text{ensmallen} \) can optionally take advantage of such attributes. In the example of a differentiable function \( f(x) \), the user can provide an implementation of the gradient \( f'(x) \), which in turn allows a first-order optimizer to be used. This generally leads to significant speedups when compared to using only \( f(x) \). To allow exploitation of such attributes, the optimizers are built to work with many types of objective functions. The classes of objective functions are listed below.

For details on the required signatures for each objective function type (such as `DifferentiableFunctionType`), see the online documentation at [https://ensmallen.org/docs.html](https://ensmallen.org/docs.html).

- **Arbitrary functions** (`ArbitraryFunctionType`). No assumptions are made on function \( f(x) \) and only the objective \( f(x) \) can be computed for a given \( x \).
- **Differentiable functions** (`DifferentiableFunctionType`). A differentiable function \( f(x) \) is an arbitrary function whose gradient \( f'(x) \) can be computed for a given \( x \), in addition to the objective.
- **Partially differentiable functions** (`PartiallyDifferentiableFunctionType`). A partially differentiable function \( f(x) \) is a differentiable function with the additional property that the gradient \( f'(x) \) can be decomposed along some basis \( j \) such that \( f'_j(x) \) is sparse. Often, this is used for coordinate descent algorithms (i.e., \( f'(x) \) can be decomposed into \( f'_1(x), f'_2(x), \) etc.).
- **Arbitrary separable functions** (`ArbitrarySeparableFunctionType`). An arbitrary separable function is an arbitrary function \( f(x) \) that can be decomposed into the sum of several objective functions: \( f(x) = \sum_i f_i(x) \).
- **Differentiable separable functions** (`DifferentiableSeparableFunctionType`). A differentiable separable function is a separable arbitrary function \( f(x) \) where the individual gradients \( f'_i(x) \) are also computable.
- **Categorical functions** (`CategoricalFunctionType`). A categorical function type is an arbitrary function \( f(x) \) where some (or all) dimensions of \( x \) take discrete values from a set.
- **Constrained functions** (`ConstrainedFunctionType`). A constrained function \( f(x) \) is a differentiable function\(^1\) subject to constraints of the form \( c_i(x) \); when the constraints are satisfied, \( c_i(x) = 0 \ \forall \ i \). Minimizing \( f(x) \) then means minimizing \( f(x) + \sum_i c_i(x) \).
- **Semidefinite programs** (SDPs). (These are a subset of constrained functions.) \( \text{ensmallen} \) has special support to make optimizing semidefinite programs [77] straightforward.

2.2 Pre-Built Optimizers

For each class of the objective functions, \( \text{ensmallen} \) provides a set of pre-built optimizers:

- **For arbitrary functions**: Simulated annealing [34], CNE (Conventional Neural Evolution) [51], DE (Differential Evolution) [73], PSO (Particle Swarm Optimization) [31], SPSA (Simultaneous Perturbation Stochastic Approximation) [71].
- **For differentiable functions**: L-BFGS [41], Frank-Wolfe [25], gradient descent.
- **For partially differentiable functions**: SCD (Stochastic Coordinate Descent) [68].
- **For arbitrary separable functions**: CMA-ES (Covariance Matrix Adaptation Evolution Strategy) [23].
- **For differentiable separable functions**: AdaBound [43], AdaDelta [83], AdaGrad [18], Adam [33], AdaMax [33], AMSBound [43], AMSGrad [60], Big Batch SGD [16], Eve [35], FTML (Follow The Moving Leader) [86], Hogwild! [59], IQN (Incremental Quasi-Newton) [50], Katyusha [6], Lookahead [84], SGD with momentum [63], Nadam [17], Nadamax [17], SGD with Nesterov momentum [52], Optimistic Adam [15], QHAdam (Quasi-Hyperbolic Optimizer) [52].

\(^1\)Generally, constrained functions do not need to be differentiable. However, this is a requirement here, as all of the current optimizers in \( \text{ensmallen} \) for constrained functions require a gradient to be available.
Adam) [44], QHSGD (Quasi-Hyperbolic Stochastic Gradient Descent) [44], RMSProp [74], SARAH/SARAH+ [53],
stochastic gradient descent, SGDR (Stochastic Gradient Descent with Restarts) [42], Snapshot SGDR [24],
SMORMS3 [21], SVRG (Stochastic Variance Reduced Gradient) [27], SWATS [32], SPAlRA (Safe Parameter-wise
Agnostic LLearning Rate Adaptation) [67], WNGrad [82].

• For categorical functions: Grid search.

• For constrained functions: Augmented Lagrangian method, primal-dual interior point SDP solver, LRSDP (low-rank
accelerated SDP solver) [11].

2.3 Example Usage

Let us consider the problem of linear regression, where we are given a matrix of predictors \( X \in \mathbb{R}^{n \times d} \) and a vector of
responses \( y \in \mathbb{R}^n \). Our task is to find the best linear model \( \theta \in \mathbb{R}^d \); that is, we want to find \( \theta^* = \text{argmin}_\theta f(\theta) \) for

\[
f(\theta) = \|X\theta - y\|^2 = (X\theta - y)^T (X\theta - y).
\]  \hspace{1cm} (2)

From this we can derive the gradient \( f'(\theta) \):

\[
f'(\theta) = 2X^T (X\theta - y).
\]  \hspace{1cm} (3)

To find \( \theta^* \) using a differentiable optimizer\(^2\), we simply need to provide implementations of \( f(\theta) \) and \( f'(\theta) \) according to
the signatures required by the DifferentiableFunctionType of objective function. For a differentiable function, only two
methods are necessary: Evaluate() and Gradient(). The pre-built L-BFGS optimizer can be used to find \( \theta^* \).

Figure 1 shows an example implementation. We hold \( X \) and \( y \) as members of the LinearRegressionFunction class,
and \( \text{theta} \) is used to represent \( \theta \). Via the use of Armadillo [65], the linear algebra expressions to implement the objective
function and gradient are readable in a way that closely matches Equations (2) and (3).

3 Advanced Usage

In addition to the support for various objective functions and the provision of many pre-built optimizers covered in Section 2,
the ensmallen library provides further functionality, suitable for advanced users. This includes:

• Inference of functions not supplied by the user. For example, a user might supply a single function named
EvaluateWithGradient() in order to allow sharing common computation that may be present in Evaluate() and
Gradient(). Those two functions can then be automatically inferred when needed.

• Callbacks, which can specify custom behavior during the optimization. Examples include printing the loss function
value at each iteration, terminating when a time limit is reached, changing the step size, and adding custom constraints
when they are violated by the current solution.

• Use of types other than arma::mat when calling Optimize(). This can include integer-valued matrices (arma::imat),
sparse matrices (arma::sp_mat), or any type whose implementation matches the Armadillo API [65, 64].

Each of the above points is covered in more detail in the following subsections.

3.1 Automatically Inferring Missing Methods

In Section 2.3, we saw an example of how a user might implement Evaluate() and Gradient() for the linear regression
objective function and use ensmallen to find the minimum. However, there is an inefficiency: the objective function
computation is defined as \( f(\theta) = \|X\theta - y\|^2 \), and the gradient computation is defined as \( f'(\theta) = 2X^T (X\theta - y) \). There is
a shared inner computation in \( f(\theta) \) and \( f'(\theta) \): the term \( (X\theta - y) \). If \( f(\theta) \) and \( f'(\theta) \) are implemented as separate functions,
there is no easy way to exploit this shared computation.

\(^2\)Typically, in practice, solving a linear regression model can be done directly by taking the pseudoinverse: \( \theta^* = X^+y \). However, the objective function
is easy to describe and useful for demonstration, so we use it in this document.
#include <ensmallen.hpp>

class LinearRegressionFunction
{
    public:
        LinearRegressionFunction(const arma::mat& in_X, const arma::vec& in_y) : X(in_X), y(in_y) { }

        double Evaluate(const arma::mat theta) { arma::vec tmp = X * theta - y; return arma::dot(tmp, tmp); }

        void Gradient(const arma::mat theta, arma::mat& gradient) { gradient = 2 * X.t() * (X * theta - y); }

    private:
        const arma::mat& X;
        const arma::vec& y;
};

int main()
{
    arma::mat X;
    arma::vec y;

    // ... set the contents of X and y here ...

    ens::LinearRegressionFunction f(X, y);

    ens::L_BFGS optimizer; // create the optimizer with default parameters

    arma::mat theta_best(X.n_rows, 1, arma::fill::randu); // initial starting point (uniform random values)

    optimizer.Optimize(f, theta_best);
    // at this point theta_best contains the best parameters
}

Figure 1: An example implementation of an objective function class for linear regression and usage of the L-BFGS optimizer in ensmallen. The online documentation for all ensmallen optimizers is at https://ensmallen.org/docs.html. The arma::mat and arma::vec types are dense matrix and vector classes from the Armadillo linear algebra library [65], with the corresponding online documentation at http://arma.sf.net/docs.html.

The differentiable optimizers in ensmallen treat the given functions as oracular, and do not know anything about the internal computations of the functions. This inefficiency\(^3\) can apply to any optimization package that accepts an objective function and its gradient as separate parameters, such as SciPy, Optim.jl and bfgsmin() from Octave [20].

To work around this issue, ensmallen internally uses template metaprogramming techniques to allow the user to provide either separate implementations of the objective function and gradient, or a combined implementation that computes both the objective function and gradient simultaneously. The latter options allows the sharing of inner computations. That is, the user can provide the methods Evaluate() and Gradient(), or EvaluateWithGradient(). For the example objective function above, we empirically show (Section 4) that the ability to provide EvaluateWithGradient() can result in a significant speedup. Similarly, when implementing a differentiable optimizer in ensmallen, it is possible to use either Evaluate() and Gradient(), or EvaluateWithGradient() during optimization.

The same technique is used to infer and provide more missing methods than just EvaluateWithGradient() or Evaluate() and Gradient(). For instance, separable functions, differentiable separable functions, constrained functions, and categorical functions each have inferrable methods. Not all of these possibilities are currently implemented in ensmallen, but the existing framework makes it straightforward to add more. Below are examples that are currently implemented:

---

\(^3\)It may be possible to use a specialized implementation of auto-differentiation or a programming language with introspection that operates directly on the abstract syntax tree [30] of the given objective and gradient computations to avoid this inefficiency and successfully share the computations across the gradient and objective. However, at the time of this writing, we are not aware of any optimization packages that actually do this.
Table 2: Available callback routines, with brief descriptions. Optional additional arguments have been omitted for brevity. See http://www.ensmallen.org/docs.html for more detailed documentation.

- (Differentiable functions.) If the user provides an objective function with Evaluate() and Gradient(), we can automatically synthesize EvaluateWithGradient().

- (Differentiable functions.) If the user provides an objective function with EvaluateWithGradient(), we can synthesize Evaluate() and/or Gradient().

- (Separable functions.) If the user provides an objective function with Evaluate() and NumFunctions(), we can synthesize a non-separable version of Evaluate().

- (Separable functions.) If the user provides an objective function with Gradient() and NumFunctions(), we can synthesize a non-separable version of Gradient().

- (Separable functions.) If the user provides an objective function with EvaluateWithGradient() and NumFunctions(), we can synthesize a non-separable version of Gradient().

For more precise details on exactly how the method generation works, see Appendix A, where we describe the framework in a simplified form, focusing only the EvaluateWithGradient()/Evaluate()/Gradient() example described above.

3.2 Callbacks

Many of the optimizers in ensmallen offer the ability to monitor and modify parts of the optimization process. Example modifications include changing the step size, adding custom constraints when they are violated by the current solution, or providing custom heuristics to find and investigate feasible solutions.

In many existing toolkits, this type of functionality is provided only via solver-specific interfaces. For instance, the tick statistical learning toolkit [9] requires the use of a solver-specific History. In contrast, ensmallen provides optimizer-independent callbacks to allow classes to inspect and work with internal parts of the optimization process. In particular, the callbacks allow code to be executed regularly during an optimization session.

RosenbrockFunction f;
arma::mat coordinates = f.GetInitialPoint();

MomentumSGD optimizer(0.01, 32, 100000, 1e-5, true, MomentumUpdate(0.5));
optimizer.Optimize(f, coordinates, EarlyStopAtMinLoss(), ProgressBar());

Figure 2: Code snippet to demonstrate usage of the callback functionality using pre-defined callbacks: EarlyStopAtMinLoss and ProgressBar. EarlyStopAtMinLoss will terminate the optimization as soon as the objective value starts increasing, and ProgressBar will print a progress bar during the optimization.
class CustomCallback
{
public:
  template<typename OptimizerType, typename FunctionType, typename MatType>
  bool Evaluate(OptimizerType & opt, FunctionType & function, const MatType & coordinates, const double objective)
  {
    std::cout << "The optimization process called Evaluate()!" << std::endl;
    return false; // Do not terminate, continue the optimization process.
  }
};

Figure 3: An example of a custom callback. This callback prints to std::cout after each time the Evaluate() function is called by the optimizer. In the example the callback always returns false, meaning that the optimization should not be terminated on behalf of the callback.

3.2.1 Using Callbacks

To use callbacks, either for optimization, tuning or logging, arbitrary callbacks to any optimizer can be optionally provided to the Optimize() function. Figure 2 contains a code snippet which briefly demonstrates usage of the callback functionality. Given the pre-defined callbacks EarlyStopAtMinLoss and ProgressBar, the code snippet shows not only how the MomentumSGD optimizer can be used to find the best coordinates but also how the callbacks can be used to control and monitor the optimization.

Eight types of optimization callback routines are available, as shown in Table 2. These callbacks are regularly called during the optimization process, depending on the objective function type. Callbacks are executed in the order that they are specified. Each callback may terminate the optimization via its bool return value, where true indicates that the optimization should be stopped. By default, subsequent callbacks are not called if an earlier callback terminates the optimization: the optimizer terminates immediately.

There are several pre-built callbacks that can be used without needing any custom code:

- EarlyStopAtMinLoss: stops the optimization process if the loss stops decreasing or no improvement has been made.
- PrintLoss: callback that prints loss to stdout or a specified output stream.
- ProgressBar: callback that prints a progress bar to stdout or a specified output stream.
- StoreBestCoordinates: callback that stores the model parameter after every epoch if the objective decreased.

Note that to use the StoreBestCoordinates callback, the user will need to instantiate a StoreBestCoordinates object, and then call BestCoordinates() in order to recover the best coordinates found during the optimization. This process is detailed more in the following section.

3.2.2 Custom Callbacks

Implementing a custom callback is straightforward: the only requirement is a class that has functions whose names are the same as the callback functions listed in Table 2. Comprehensive documentation on the required signatures for each callback can be found in the ensmallen documentation at http://ensmallen.org/docs.html#callback-states.

If a custom callback is desired to take an action before the optimization process starts, then only a CustomCallback::BeginOptimization() method is required to perform the desired action. An example of a custom callback that prints a line to std::cout every time the optimization calls Evaluate() is shown in Figure 3. The CustomCallback class can be used exactly like EarlyStopAtMinLoss in Figure 2—the only change needed is to add CustomCallback() to the list of arguments passed to optimizer.Optimize().

If a callback class requires additional parameters or state beyond what is passed through the predefined arguments to functions in Table 2, a user should manually create an instance of the callback class with those additional parameters, and then pass the object to the optimizer as a callback when Optimize() is called. ensmallen does not modify or dereference the object, so it is safe to use for this purpose. Figure 4 provides an example, where we pass an instantiated custom callback that takes an additional step-size decay parameter as input. In addition, inside the StepTaken() callback, we use the optimizer’s
```cpp
struct CustomCallback
{
    CustomCallback(double rIn) : r(rIn) {}

    template<typename OptimizerType, typename FunctionType, typename MatType>
    void StepTaken(OptimizerType & optimizer, FunctionType & function, MatType & coordinates)
    {
        // Multiply the step size by r (hopefully r is less than 1!).
        optimizer.StepSize() *= r;
    }

double r;
};
```

```cpp
RosenbrockFunction f;
arma::mat coordinates = f.GetInitialPoint();

Adam opt;
CustomCallback cb(0.9); // Instantiate the custom callback...
opt.Optimize(f, coordinates, cb); // ...and call Optimize() with that object!
```

**Figure 4:** Code snippet demonstrating how to add additional parameters/state to a callback and accessing optimizer-specific parameters.

```cpp
struct Optimizer
{
    template<typename FT>
    void Optimize(FT & f, arma::mat & p)
    {
        f.Evaluate(p);
    }
};
```

```cpp
int main()
{
    RosenbrockFunction rf;
    arma::mat parameters = rf.GetInitialPoint();
    Optimizer opt;
    opt.Optimize(rf, parameters);
    return 0;
}
```

```cpp
struct Optimizer
{
    template<typename FT, typename... CallbackType>
    void Optimize(FT & f, arma::mat & p, CallbackType && ... c)
    {
        Callback::BeginOptimization(*this, f, p, c...);
        f.Evaluate(iterate);
    }
};
```

```cpp
int main()
{
    RosenbrockFunction rf;
    arma::mat parameters = rf.GetInitialPoint();
    Optimizer opt;
    opt.Optimize(rf, parameters);
    return 0;
}
```

**Figure 5:** Left panel: A C++ program that mimics the ensmallen optimizer interface without any callback functionality. Right panel: A corresponding C++ program using an empty callback routine that is automatically optimized out. Both programs produce the exact same machine code, resulting in no performance penalty if no callbacks are used.

there is no performance penalty if no callbacks are used. Figure 5 shows two programs; one explicitly without callbacks, and one with an empty callback. The implementation of callback facility is internally done via template metaprogramming. In all cases, modern C++ compilers (e.g. clang-1100.0.33.16 and g++ 9.2.1) optimized away the unused code; the resultant machine code appears as if the callback code never existed in the first place. Both programs produce the exact same machine code, indicating that there is no performance penalty if no callbacks are used.

interface function (StepSize()) to update the step size. Note that if this callback is attempted to be used with an optimizer that did not have a StepSize() function, compilation would fail. As such, some care is necessary when implementing custom callbacks.

There is no performance penalty if no callbacks are used. Figure 5 shows two programs; one explicitly without callbacks, and one with an empty callback. The implementation of callback facility is internally done via template metaprogramming. In all cases, modern C++ compilers (e.g. clang-1100.0.33.16 and g++ 9.2.1) optimized away the unused code; the resultant machine code appears as if the callback code never existed in the first place. Both programs produce the exact same machine code, indicating that there is no performance penalty if no callbacks are used.
Details on the internal implementation of the callback system can be found in Appendix B.

### 3.3 Optimization With Various Matrix Types

In the example shown in Section 2.3, where we introduced the class `LinearRegressionFunction`, the matrix and vector objects are hardcoded as `arma::mat` and `arma::vec`. These objects hold elements with the C++ type double, representing double-precision floating point [22]. However, in many applications it can be very important to specify a different underlying element type (e.g. the single-precision `float`, used by `arma::fmat` and `arma::fvec`). For instance, in the field of machine learning, neural networks have been shown to be effective with low-precision floating point representations for weights [79]. Furthermore, many optimization problems have parameters that are best represented as sparse data [75, 59], which is represented in Armadillo as the `sp_mat` class [64, 66]. Even alternate representations such as data held on the GPU can be quite important: the use of GPUs can often result in significant speedups [55, 8].

In order to handle this diverse set of needs, ensmallen has been built in such a way that any underlying storage type can be used to represent the coordinates to be optimized—so long as it matches the Armadillo API. This means that the Bandicoot GPU matrix library can be used as a drop-in replacement once it is stable.

An example of seamlessly using ensmallen’s optimizers with different underlying storage types is given in Figure 6. In this example, the types `arma::mat` (which holds double), `arma::fmat` (which holds float), and `arma::imat` (which holds integers) can be used.

```cpp
#include <ensmallen.hpp>

// A trivial parabolic function. The minimum is at the origin. The function
// works with any matrix type.

class ParabolicFunction
{
  public:
    template<typename MatType>
    typename MatType::elem_type Evaluate(const MatType& coordinates)
    {
      // Return the sum of squared coordinates.
      return arma::accu(arma::square(coordinates));
    }
};

int main()
{
  // Use simulated annealing to optimize the ParabolicFunction.
  ParabolicFunction pf;
  ens::SA<> optimizer(ens::ExponentialSchedule);

  // element type is double
  arma::mat doubleCoordinates(10, 1, arma::fill::randu);
  optimizer.Optimize(pf, doubleCoordinates);

  // element type is float
  arma::fmat floatCoordinates(10, 1, arma::fill::randu);
  optimizer.Optimize(pf, floatCoordinates);

  // element type is int
  arma::imat intCoordinates(10, 1, arma::fill::randi);
  optimizer.Optimize(pf, intCoordinates);
}
```

Figure 6: Example ensmallen program showing that ensmallen’s optimizers can be used to seamlessly optimize functions with many different element types.

---

#include <ensmallen.hpp>

// A trivial parabolic function. The minimum is at the origin. The function
// works with any matrix type.

class ParabolicFunction
{
  public:
    template<typename MatType>
    typename MatType::elem_type Evaluate(const MatType& coordinates)
    {
      // Return the sum of squared coordinates.
      return arma::accu(arma::square(coordinates));
    }
};

int main()
{
  // Use simulated annealing to optimize the ParabolicFunction.
  ParabolicFunction pf;
  ens::SA<> optimizer(ens::ExponentialSchedule);

  // element type is double
  arma::mat doubleCoordinates(10, 1, arma::fill::randu);
  optimizer.Optimize(pf, doubleCoordinates);

  // element type is float
  arma::fmat floatCoordinates(10, 1, arma::fill::randu);
  optimizer.Optimize(pf, floatCoordinates);

  // element type is int
  arma::imat intCoordinates(10, 1, arma::fill::randi);
  optimizer.Optimize(pf, intCoordinates);
}
```cpp
class GradientDescent
{
    template<typename FunctionType, typename MatType, typename GradType = MatType>
    typename MatType::elem_type Optimize(FunctionType& function, MatType& coordinates)
    {
        // The step size is hardcoded to 0.01, and the number of iterations is 1000.
        for (size_t i = 0; i < 1000; ++i)
        {
            GradType gradient;
            function.Gradient(coordinates, gradient);

            // Take the step.
            coordinates -= 0.01 * gradient;
        }

        // Compute and return the final objective.
        return function.Evaluate(coordinates);
    }
};
```

Figure 7: Example implementation of a simple gradient descent optimizer. For the sake of brevity, functionality such as the ability to configure the parameters has been deliberately omitted. The actual GradientDescent optimizer in ensmallen provides more functionality.

int) are all used with ensmallen’s simulated annealing implementation to optimize a very simple parabolic function. Due to the use of templates both inside ensmallen and in the implementation of the ParabolicFunction class, it is trivial to change the underlying type used for storage.

Consider the simplified gradient descent optimizer shown in Figure 7. The use of the template types FunctionType, MatType, and GradType means that at compilation time, the correct types are substituted in for FunctionType, MatType, and GradType (which by default is set to be the same as MatType in this code). So long as each type has all the methods that are used inside of Optimize(), there will be no compilation problems. Templates are a technique for code generation; in this case, that means the code generated will be exactly the same as if Optimize() was written with the types specified in those template parameters. This means that there is no additional runtime overhead when a different MatType is used.

Appendix C contains details on the internal compile-time checks used for providing users with concise error messages, avoiding the long errors typically associated with template metaprogramming.

4 Experiments

To show the efficiency of mathematical optimization with ensmallen, we compare its performance with several other commonly used optimization frameworks, including some that use automatic differentiation.

4.1 Simple Optimizations and Overhead

For our first experiment, we aim to capture the overhead involved in various optimization toolkits. In order to do this, we consider the simple and popular Rosenbrock function [62]:

\[
f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1^2). \tag{4}\]

This objective function is useful for this task because the computational effort involved in computing \( f(\cdot) \) is trivial. Therefore, if we hold the number of iterations of each toolkit constant, then this will help us understand the overhead costs of each toolkit. For the optimizer, we use simulated annealing [34], a gradient-free optimizer. Simulated annealing will call the objective function numerous times; for each simulation we limit the optimizer to 100K objective evaluations.

The code used to run this simulation for ensmallen (including the implementation of the Rosenbrock function) is given in Figure 8. Note that the RosenbrockFunction is actually implemented in ensmallen’s source code, in the directory include/ensmallen_bits/problems/.
```cpp
#include <ensmallen.hpp>

struct RosenbrockFunction
{
    template <typename MatType>
    static typename MatType::elem_type Evaluate(const MatType& x) const
    {
        return 100 * std::pow(x[1] - std::pow(x[0], 2), 2) + std::pow(1 - x[0], 2);
    }
};

int main()
{
    arma::wall_clock clock;

    RosenbrockFunction rf;
    ens::ExponentialSchedule sched;
    // A tolerance of 0.0 means the optimization will run for the maximum number of iterations.
    ens::SA<> s(sched, 100000, 10000, 1000, 100, 0.0);

    // Get the initial point of the optimization.
    arma::mat parameters = rf.GetInitialPoint();

    // Run the optimization and time it.
    clock.tic();
    s.Optimize(rf, parameters);
    const double time = clock.toc();
    std::cout << time << std::endl << "Result (optimal 1, 1): " << parameters.t();
    return 0;
}
```

Figure 8: Code to use ensmallen to optimize the Rosenbrock function using 100K iterations of simulated annealing.

We compare four frameworks for this task:

(i) ensmallen,
(ii) scipy.optimize.anneal from SciPy 0.14.1 [29],
(iii) simulated annealing implementation in Optim.jl with Julia 1.0.1 [49],
(iv) samin in the optim package for Octave [20].

While another option here might be simulannealbnd() in the Global Optimization Toolkit for MATLAB, no license was available. We ran our code on a MacBook Pro i7 2018 with 16GB RAM running macOS 10.14 with clang 1000.10.44.2, Julia version 1.0.1, Python 2.7.15, and Octave 4.4.1.

Our initial implementation for each toolkit, corresponding to the line “default” in Table 3, was as simple of an implementation as possible and included no tuning. This reflects how a typical user might interact with a given framework. Only Julia and ensmallen are compiled, and thus are able to avoid the function pointer dereference for evaluating the Rosenbrock function and take advantage of inlining and related optimizations. The overhead of both scipy and samin is quite large—ensmallen is nearly three orders of magnitude faster for the same task.

However, both Python and Octave have routes for acceleration, such as Numba [37], MEX bindings and JIT compilation. We hand-optimized the Rosenbrock implementation using Numba, which required significant modification of the underlying anneal.anneal() function. These techniques did produce some speedup, as shown in the second row of Table 3. For Octave, a MEX binding did not produce a noticeable difference. We were unable to tune either ensmallen or Optim.jl to get any speedup, suggesting that novice users will easily be able to write efficient code in these cases.
4.2 Large-Scale Linear Regression Problems

Next, we consider the linear regression example described in Section 2.3. For this task we use the first-order L-BFGS optimizer [41], implemented in ensmallen as the L_BFGS class. Using the same four packages, we implement the linear regression objective and gradient. Remembering that ensmallen allows us to share work across the objective function and gradient implementations (Section 3.1), for ensmallen we implement a version with only EvaluateWithGradient(), denoted as ensmallen-1. We also implement a version with both Evaluate() and Gradient(): ensmallen-2. We also use automatic differentiation for Julia via the ForwardDiff.jl [61] package and for Python via the Autograd [45] package. For GNU Octave we use the bfgsmin() function.

Results for various data sizes are shown in Table 4. For each implementation, L-BFGS was allowed to run for only 10 iterations and never converged in fewer iterations. The datasets used for training are highly noisy random data with a slight linear pattern. Note that the exact data is not relevant for the experiments here, only its size. Runtimes are reported as the average across 10 runs.

The results indicate that ensmallen with EvaluateWithGradient() is the fastest approach. Furthermore, the use of EvaluateWithGradient() yields non-negligible speedup over the ensmallen-2 implementation with both the objective and gradient implemented separately. In addition, although the automatic differentiation support makes it easier for users to write their code (since they do not have to write an implementation of the gradient), we observe that the output of automatic differentiators is not as efficient, especially with ForwardDiff.jl. We expect this effect to be more pronounced with increasingly complex objective functions.

4.3 Easy Pluggability of Various Optimizers

Lastly, we demonstrate the easy pluggability in ensmallen for using various optimizers on the same task. Using a version of LinearRegressionFunction from Section 2.3 adapted for separable differentiable optimizers, we run six optimizers with default parameters in just 8 lines of code, as shown in Fig. 9. Applying these optimizers to the YearPredictionMSD dataset from the UCI repository [40] yields the learning curves shown in Fig. 10.

```// X and y are data. LinearRegressionFunction lrf(X, y);

using namespace ens;
StandardSGD<>().Optimize(lrf, sgdModel);
Adam().Optimize(lrf, adamModel);
AdaGrad().Optimize(lrf, adagradModel);
SMORMS3().Optimize(lrf, smorms3Model);
SPALeRASGD().Optimize(lrf, spaleraModel);
RMSProp().Optimize(lrf, rmspropModel);
```

Figure 9: ensmallen makes it easy to switch out optimizer types: 8 lines of code run 6 optimizers on one problem.

Any other optimizer for separable differentiable objective functions can be dropped into place in the same manner; given the large number of available optimizers in ensmallen, this support could be used to easily compare optimizers. In fact, this is exactly done with the interactive optimizer visualization tool found at https://vis.ensmallen.org. Figure 11 shows an example visualization.

<table>
<thead>
<tr>
<th></th>
<th>ensmallen</th>
<th>scipy</th>
<th>Optim.jl</th>
<th>samin</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>0.004s</td>
<td>1.069s</td>
<td>0.021s</td>
<td>3.173s</td>
</tr>
<tr>
<td>tuned</td>
<td>0.574s</td>
<td></td>
<td></td>
<td>3.122s</td>
</tr>
</tbody>
</table>

Table 3: Runtimes for 100K iterations of simulated annealing with various frameworks on the simple Rosenbrock function. Julia code runs do not count compilation time. The tuned row indicates that the code was manually modified for speed.
Table 4: Runtimes for the linear regression function on various dataset sizes, with $n$ indicating the number of samples, and $d$ indicating the dimensionality of each sample. All Julia runs do not count compilation time.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>$d$: 100, $n$: 1k</th>
<th>$d$: 100, $n$: 10k</th>
<th>$d$: 100, $n$: 100k</th>
<th>$d$: 1k, $n$: 100k</th>
</tr>
</thead>
<tbody>
<tr>
<td>ensmallen-1</td>
<td>0.001s</td>
<td>0.009s</td>
<td>0.154s</td>
<td>2.215s</td>
</tr>
<tr>
<td>ensmallen-2</td>
<td>0.002s</td>
<td>0.016s</td>
<td>0.182s</td>
<td>2.522s</td>
</tr>
<tr>
<td>Optim.jl</td>
<td>0.006s</td>
<td>0.030s</td>
<td>0.337s</td>
<td>4.271s</td>
</tr>
<tr>
<td>scipy</td>
<td>0.003s</td>
<td>0.017s</td>
<td>0.202s</td>
<td>2.729s</td>
</tr>
<tr>
<td>bfgsmin</td>
<td>0.071s</td>
<td>0.859s</td>
<td>23.220s</td>
<td>2859.81s</td>
</tr>
<tr>
<td>ForwardDiff.jl</td>
<td>0.497s</td>
<td>1.159s</td>
<td>4.996s</td>
<td>603.106s</td>
</tr>
<tr>
<td>autograd</td>
<td>0.007s</td>
<td>0.026s</td>
<td>0.210s</td>
<td>2.673s</td>
</tr>
</tbody>
</table>

Figure 10: Example usage of six ensmallen optimizers to optimize a linear regression function on the YearPredictionMSD dataset \cite{40} for 5 epochs of training. The optimizers can be tuned for better performance.
5 Conclusion

This report introduces and explains ensmallen, a C++ mathematical optimization library that internally uses template metaprogramming to produce efficient code. The library is flexible, with support for numerous types of objective functions, and many implemented optimizers. It is easy to both implement objective functions to optimize with ensmallen, and to write new optimizers for inclusion in the library. ensmallen has support for automatic function inference and callbacks. Because this is done through template metaprogramming, there is no additional runtime overhead. Empirical results show that ensmallen outperforms other toolkits for similar tasks.

Future work includes the implementation of additional optimizers and better support for various types of objective functions (such as better support for constrained functions). Development is done in an open manner at https://github.com/ensmallen/ensmallen. The library uses the permissive BSD license [72]. Anyone is welcome to help with the effort and contribute code.
Appendix A: Inferring Missing Methods – Details

In Section 3.1, we described that ensmallen is able to infer missing methods. For instance, if the user provides a differentiable objective function that implements EvaluateWithGradient(), we can automatically infer and provide implementations of Evaluate() and Gradient() that the optimizer can then use. (And vice versa: we can infer EvaluateWithGradient() from Evaluate() and Gradient().)

Here, we provide details of how this is implemented, focusing specifically on how EvaluateWithGradient() can be inferred, and how this is used in ensmallen’s optimizers. The same technique is applicable to all of the other methods that ensmallen infers.

Using our framework for function inference is straightforward; an example differentiable optimizer may resemble the code in Figure 12. Code for a new optimizer only needs to define a Function<FunctionType, ...> wrapper at the beginning of optimization, and can then expect all three of Evaluate(), Gradient(), and EvaluateWithGradient() to be available. The class Function<...> is like a mix-in class [69]; more accurately, it is a collection of mix-in classes.

Figure 13 shows a shortened snippet of the definition of the Function class. We can see that Function inherits methods from the given FunctionType (which is the user-supplied objective function class that is to be optimized), and also from the AddEvaluateWithGradient<...> mixin class. The key to our approach is that if FunctionType class has an EvaluateWithGradient() method, then AddEvaluateWithGradient<...> will provide no functions; if FunctionType does not have an EvaluateWithGradient() method, then AddEvaluateWithGradient<...> will provide an EvaluateWithGradient() function.

The details of how this work depend on the SFINAE technique [2, 78] and template specialization. The AddEvaluateWithGradient class has five template parameters, shown in Figure 14; the last two of these have default values.

For the sake of brevity we omit the expressions for the bool parameters HasEvaluateGradient and HasEvaluateWithGradient. These two parameters are traits expressions that depend heavily on SFINAE techniques for

```cpp
// FunctionType is the user-supplied function type to optimize.
// MatType is the user-supplied type of the initial coordinates.
// GradType is the user-specified type of the gradient.
// typename MatType::elem_type represents the internal type held by MatType
// (e.g., if MatType is arma::mat, then this is double)
template<typename FunctionType, typename MatType, typename GradType>
typename MatType::elem_type Optimize(FunctionType& function,
                                       MatType& coordinates)
{
    // The Function<> mix-in class adds all inferrable methods to `function`
    // So, if `function` has `function.Evaluate()` and `function.Gradient()`, then
    // `fullFunction` will have `fullFunction.EvaluateWithGradient()`.
    typename FunctionType<FunctionType, MatType, GradType> fullFunction(function);

    // The rest of the optimizer's code should use `fullFunction`, not `function`.
    ...
}
```

Figure 12: An example implementation of an ensmallen optimizer’s Optimize() method. The use of the Function<> mix-in class [69] will provide methods that the original FunctionType may not have.

```cpp
template<typename FunctionType, typename MatType, typename GradType>
class Function :
    public AddEvaluateWithGradient<FunctionType, MatType, GradType>,
    public FunctionType
    ... // many other mixin classes omitted
```

Figure 13: Snippet of the definition of the Function class. The full implementation contains numerous other mix-in classes like AddEvaluateWithGradient.
method detection; HasEvaluateGradient will evaluate to true if FunctionType has Evaluate() and Gradient(); if only one (or neither) is available, the value is false. HasEvaluateWithGradient will evaluate to true if FunctionType has an overload of EvaluateWithGradient(), and false otherwise.

Using these two boolean template variables, we can then use template specialization to control the behavior of AddEvaluateWithGradient as a function of what is provided by FunctionType. Specifically, we make two specializations: one for when EvaluateWithGradient() exists, and one for when both Evaluate() and Gradient() exist but EvaluateWithGradient() does not.

The first specialization, shown in Figure 15, is for when HasEvaluateWithGradient is true—meaning that FunctionType already has EvaluateWithGradient(). In this situation, the implementation of

```
template<typename FunctionType,
          typename MatType,
          typename GradType,
          bool HasEvaluateGradient>
class AddEvaluateWithGradient
{
    public:
    // Reflect the existing EvaluateWithGradient().
    typename MatType::elem_type EvaluateWithGradient(
        const MatType& coordinates, GradType& gradient)
    {
        return static_cast<
            typename FunctionType>
            (<
                static_cast<typename FunctionType>
                (<
                    static_cast<brayuk2022@brayuk2022.com>
                ,
                MatType,
                GradType>>)(
                    this))
            ->EvaluateWithGradient(
                coordinates, gradient);
    }
};
```

Figure 15: Partial template specialization of AddEvaluateWithGradient for when the given FunctionType does have an EvaluateWithGradient() method available already.
AddEvaluateWithGradient:
EvaluateWithGradient() simply calls out to FunctionType::EvaluateWithGradient(). However, there is some complexity here, as to successfully call FunctionType::EvaluateWithGradient(), we must first cast the this pointer to have type FunctionType—which can only be done by first casting this to the derived class Function<...>. (This cast is only safe because we know that we will never create an AddEvaluateWithGradient outside of the context of the Function<> class.)

Next, we specialize for the case where HasEvaluateGradient is true (i.e., FunctionType has both Evaluate() and Gradient() methods), and HasEvaluateWithGradient is false (i.e., there is no EvaluateWithGradient() provided by FunctionType). In this situation, we intend to synthesize an implementation for EvaluateWithGradient() by using both of the provided Evaluate() and Gradient() methods sequentially. Figure 16 shows this specialization.

The same complexity with casting is necessary in this specialization too. The last specialization is the general case: where there is neither Evaluate() and Gradient() nor EvaluateWithGradient() provided by FunctionType. That case is covered by the initial shown implementation of AddEvaluateWithGradient: EvaluateWithGradient() is provided with an argument of type traits::UnconstructableType—which is just a class with a non-public constructor, that cannot be created; and thus, this function cannot be called. It is necessary to have this unusable version of EvaluateWithGradient(), though, because the design pattern requires that AddEvaluateWithGradient always provides a method with the name EvaluateWithGradient().

This does, however, mean that users can get long and confusing error messages if the optimizer attempts to instantiate the overload of EvaluateWithGradient() with traits::UnconstructableType. But, remember that this situation can only happen when using a differentiable optimizer if the user provided a FunctionType that (a) does not have both Evaluate() or Gradient(), or (b) does not have EvaluateWithGradient(). This is something that we already have code to detect—that is the code that computes the values of the boolean template parameters HasEvaluateGradient and HasEvaluateWithGradient. Therefore, we simply use a static_assert() to provide a clear and understandable error message at compile time when neither of those values are true. We encapsulate this in a convenient function that can be added at the beginning of the Optimize() method:

    CheckFunctionTypeAPI<FunctionType, MatType, GradType>();

This technique is adapted to each objective function type that ensmallen supports, in order to provide straightforward error messages of the form shown in Figure 17. This static_assert() failure output appears typically at the end of the error output, which is far preferable to the output without CheckFunctionTypeAPI. Example output produced by clang++ is shown in Figure 18.

Note that the error message above can be highly misleading: it’s not actually a requirement that EvaluateWithGradient() be supplied by the objective function class! That can be automatically inferred, just like discussed above, but only if

template<typename FunctionType, typename MatType, typename GradType>
class AddEvaluateWithGradient<FunctionType, MatType, GradType, true, false>
{
  public:
    // Use FunctionType's Evaluate() and Gradient().
    typename MatType::elem_type EvaluateWithGradient(const MatType& coordinates, GradType& gradient)
    {
        const typename MatType::elem_type objective =
            static_cast<Function<FunctionType,
                MatType, GradType>>(this)->Evaluate(coordinates);

        static_cast<Function<FunctionType,
            MatType, GradType>>(this)->Gradient(coordinates, gradient);

        return objective;
    }
};

Figure 16: Partial template specialization of AddEvaluateWithGradient for when EvaluateWithGradient() is not available, but Evaluate() and Gradient() are.
Figure 17: Example error output produced by clang++ when the given objective function is missing methods and static_assert()s are used. Compare with Figure 18, which is far less clear.

Figure 18: Example error output produced by clang++ when the given objective function is missing methods, and static_assert()s are not used. Compare with Figure 17, which is much clearer.

Evaluate() and Gradient() are both available. In the case of the example code that generated the errors above, Gradient() was not implemented—thus, the given error message is actually somewhat inaccurate and confusing!

The last piece of the puzzle is making sure that when Function<...>::EvaluateWithGradient() is called, that this always calls AddEvaluateWithGradient::EvaluateWithGradient(). This can be done by a simple using declaration in the body of the Function class:

```cpp
using AddEvaluateWithGradient<FunctionType,
  MatType,
  GradType>::EvaluateWithGradient;
```

This is all that is needed to infer and synthesize an EvaluateWithGradient() method when the user provided an objective function that only has Evaluate() and Gradient(). There is one detail we have omitted, though—the code that we have shown here handles non-const and non-static versions of methods provided by the user. Separate auxiliary structures like AddEvaluateWithGradientConst and AddEvaluateWithGradientStatic are used for function inference in those cases; the general design is identical.

All of these pieces put together result in a clean interface that inference of methods that users did not provide in their objective function. Further, this all happens at compile time and thus there is no runtime penalty. Auxiliary structures like AddEvaluateWithGradient should be optimized out by the compiler.

All of the code involved with automatic generation of missing methods can be found in the ensmallen source code in the
Appendix B: Callbacks – Details

In Section 3.2, we introduced ensmallen’s support for callbacks. This support is implemented via templates—like much of the rest of ensmallen. This design strategy was chosen in order to minimize runtime overhead caused by callbacks, and produce machine code roughly equivalent to the machine code that would have been produced if the callback had been directly integrated into the optimizer’s code.

In this section, we detail the implementation of these callbacks and show how to implement optimizers that are capable of handling these callbacks.

An optimizer that supports callbacks should call out to the static methods in the Callback class inside of its Optimize() method, as below:

```cpp
double functionValue = function.Evaluate(coordinates);
bool terminate = Callback::Evaluate(*this, function, coordinates, functionValue, callbacks...);
```

In the above snippet, function represents the function being optimized, *this is the optimizer itself, coordinates represents the current coordinates of the optimization, and callbacks is a template vararg pack containing all of the given callbacks. Note that this means callbacks can be empty.

The Callback class contains a top-level method to use for each of the callback types supported by ensmallen, as listed in Table 2. There is a Callback::Gradient() method, Callback::EvaluateConstraint(), Callback::BeginEpoch(), and so forth.

The Callback::Evaluate() method calls the Evaluate() method of every given callback that has an Evaluate() method implemented. This means that there is some amount of difficulty we have to handle: not every callback in callbacks... will have an Evaluate() method available. Thus, we can use the same techniques as in Section 3.1 to detect, via SFINAE, whether an Evaluate() method exists for a given callback, and then perform the correct action based on the result.

The definition of Callback::Evaluate() is given in Figure 19. Each callback function takes several template parameters, including OptimizerType (the type of optimizer that is being used), FunctionType (the type of function being optimized), MatType (the matrix type used to store the coordinates), and CallbackTypes (the set of types of callbacks). Of these, CallbackTypes is the most important. Since it is a template vararg, the type CallbackTypes is actually a pack that corresponds to every type of every callback that must be called.

The implementation of the function unpacks the callbacks, calling each callback’s Evaluate() method (if it exists) in sequence, and terminating early if any of these callbacks returns true. Each callback’s Evaluate() method is called through the helper function Callback::EvaluateFunction(), which uses SFINAE traits to control behavior depending on whether the given callback has an Evaluate() method or not. Figure B shows the two overloads of Callback::EvaluateFunction().

```cpp
template<typename OptimizerType, typename FunctionType, typename MatType, typename... CallbackTypes>
static bool Callback::Evaluate(OptimizerType& optimizer,
                                FunctionType& function,
                                const MatType& coordinates,
                                const double objective,
                                CallbackTypes&... callbacks)
{
    // This will return immediately once a callback returns true.
    bool result = false;
    (void) std::initializer_list<bool>{ result =
        result || Callback::EvaluateFunction(callbacks, optimizer, function, coordinates, objective)... };
    return result;
}
```

Figure 19: Implementation of Callback::Evaluate(), showing part of the process of calling each callback given in the template vararg pack callbacks.
Figure 20: Implementation of Callback::EvaluateFunction(). Two overloads are specified: the first is used when CallbackType has an Evaluate() method, and the second is used when CallbackType does not have an Evaluate() method. SFINAE is used, via callbacks::traits::HasEvaluateSignature, to determine which overload to use.

Similar to Section 3.1, a traits class is used to determine which of the two overloads of EvaluateFunction() to call. The traits class callbacks::traits::HasEvaluateSignature<...>::value evaluates to true only when the inner callback and arguments can form a valid Evaluate() signature. Thus, the correct overload is called when EvaluateFunction() is called from Callback::Evaluate().

Each of the other callbacks in the Callback class is implemented in virtually identical form, although with different arguments depending on what the callback is. For further details on the callback infrastructure, examples are provided in the code repository in the directory include/ensmallen_bits/callbacks/.

Appendix C: Static Assertions to Check Matrix Traits

In Section 3.3, we showed that ensmallen uses templates to allow arbitrary types to be used for the matrix type or gradient type. However, care must be taken there to avoid a drawback with C++ template metaprogramming, which is the issue of compiler errors. In Figure 7, the compiler may fail to substitute a given MatType into the code for GradientDescent::Optimize(). A typical reason might be that a required method of MatType or FunctionType is not available. For example, the given MatType does not have an operator() method. Ordinarily, current C++ compilers typically emit a long list of error messages, with unnecessarily detailed and distracting information. The internal framework in ensmallen avoids this issue via the use of C++ compile-time static assertions (static assert()), resulting in considerably clearer error messages.

The framework provides a set of utility methods for checking traits of MatType and GradType (for differentiable objective functions) inside of an optimizer's Optimize() method:

- RequireFloatingPointType<MatType>(): this requires that the element type held by MatType is either float or double. This is generally needed by optimizers that use Armadillo functionality which ends up calling LAPACK functions [7].

- RequireSameInternalTypes<Mat1Type, Mat2Type>(): this requires that Mat1Type and Mat2Type use the same type to hold elements. This is useful for differentiable optimizers, where the user is allowed to specify different objective matrix types (MatType) and gradient matrix types (GradType). This function allows us to require that the same type (e.g., int or float or double) is used for both MatType and GradType.
/**
 * Require that the internal element type of the matrix type and gradient type
 * are the same. A static_assert() will fail if not.
 */

template<
    typename MatType,     
    typename GradType>
void RequireSameInternalTypes()
{
#ifndef ENS_DISABLE_TYPE_CHECKS
    static_assert(
        std::is_same<
            typename MatType::elem_type,
            typename GradType::elem_type>::value,
        "The internal element types of the given MatType and GradType must be 
        identical, or it is not known to work! If you would like to try 
        anyway, set the preprocessor macro ENS_DISABLE_TYPE_CHECKS before 
        including ensmallen.hpp. However, you get to pick up all the pieces if 
        there is a failure!");
#endif
}

Figure 21: Implementation of RequireSameInternalTypes() from internal ensmallen code.

- RequireDenseFloatingPointType<MatType>(): this requires that the element type held by MatType is either float or double, and that the representation used for storage by MatType is dense.

These methods would typically be called at the top of the implementation of an optimizer’s Optimize() method. At the time of writing, these are the only three checks that have been needed, but it is easy to add more. The implementation of these functions is just a static_assert() that uses some underlying traits of the given template parameters. For example, Figure 21 is the implementation of RequireSameInternalTypes<...>().

Note that users who would like to continue despite these static_assert() checks failing may simply define the macro ENS_DISABLE_TYPE_CHECKS in their code before the line #include <ensmallen.hpp>.
References


