POLO Documentation

Arda Aytekin **Martin Biel** **Mikael Johansson**

Jun 27, 2019

CONTENTS

`polo` is a header-only C++ library intended to help researchers and practitioners solve optimization problems of the form

$$\underset{x\in\mathbb{R}^d}{\text{minimize}} \quad \phi(x) := \mathrm{f}(x) + \mathrm{h}(x) \tag{1}$$

on different computing platforms. Here, the total loss consists of two parts. The first part, $\mathrm{f}(\cdot)$, is a *smooth* function of the d-dimensional decision vector, x, and it consists of N *component* functions:

$$\mathrm{f}(x) = \sum_{n=1}^{N} \mathrm{f_n}(x)\,.$$

The second part, $\mathrm{h}(\cdot)$, is a possibly *nonsmooth* function of the decision vector, x.

In this documentation, we aim at providing the necessary information for both users and future contributors to get started with `polo`. In *Getting Started*, we show how to install the library and what a typical workflow involving `polo` looks like. Then, in three parts, we cover different aspects of the library.

In the first part, we follow a tutorial-based approach to show users how to use `polo` for solving Problem (1) on different computing platforms. In *Serial Execution*, we focus on building *proximal gradient algorithms* that run *sequentially* on a single CPU. Then, in *Shared-Memory Execution*, we switch to parallel versions of the proximal gradient algorithms, where all the component functions and the decision vector reside in a *shared* memory space. These algorithms benefit from multiple CPUs to speed up gradient computations of the smooth loss. Finally, in *Distributed-Memory Execution*, we focus on solving Problem (1) when the component functions and the decision vector are *distributed* among different nodes.

The second part of the documentation covers more advanced topics that are mostly related to the internals of the library. These topics would serve as a starting point for users and contributors to extend the functionalities of the library. In *Proximal Gradient Methods*, we cover the abstraction for the proximal gradient algorithms, and provide detailed information on their *policies*. Then, in *Utilities*, we cover the functionalities provided by the `utilities` layer of the library. Finally, in *C-API*, we provide the C API of the library, which implements many algorithms from the proximal gradient family and can be used from high-level languages to solve Problem (1).

In the last part, we provide high-level language integrations of `polo`. Currently, `polo` is wrapped and extended in the Julia language, and we will cover the library in *POLO.jl*.

CHAPTER

ONE

GETTING STARTED

In this chapter, we focus on how to get started with `polo`.

1.1 Setting Up

In this section, we will cover how to download, and install `polo` from source, or use the prebuilt multi-architecture Docker images. The commands given in this section are valid `bash` commands, and they should work on Unix-like systems as well as Windows[1].

1.1.1 Obtaining the Library

`polo` is hosted at GitHub under the permissive MIT license. There are two different ways to obtain the library.

The first, and the suggested, way is to use git to clone the repository, and check out a specific version (i.e., a snapshot). For instance, the following code will clone the latest snapshot (i.e., the `master` branch) of the library

```
git clone https://github.com/pologrp/polo $HOME/polo
```

to `polo` directory under `$HOME`, which serves as the default repository for a user's personal files on Unix-like systems. Later, we can change directory and check out a different snapshot by issuing the following command

```
cd $HOME/polo
git checkout v1.0.0
```

Here, we are checking out the `v1.0.0` snapshot.

The second way is to `download` the `master` branch as a zip file (named, e.g., `polo-master.zip` under `$HOME/Downloads` directory), and unzip its contents under `$HOME` by issuing the following command

```
unzip $HOME/Downloads/polo-master.zip -d $HOME
cd $HOME/polo-master # changes directory to the unzipped library
```

Similarly, we can visit the releases page of the library, download the specific snapshots, and follow the same procedure to unzip their contents.

Note: Currently, there are no tagged versions/snapshots in the library. Hence, throughout the documentation, we assume that the latest snapshot (i.e., the `master` branch) is checked out under `$HOME/polo` directory.

[1] Windows users *might* need to install Cygwin or Windows Subsystem for Linux.

Todo: After tagging the first version, reword the above note.

1.1.2 Installing the Library

`polo` is a C++ template library that uses C++11 features. As a result, we need to have a compiler that supports these features. Among the well-known compilers, `gcc v4.8.1`, `clang v3.3`, and `Visual Studio 2015`, together with their newer versions, all support C++11.

In addition to the C++ feature dependencies, `polo` requires CMake (at least `v3.9.0`) to manage its *optional* dependencies:

1. Reference or optimized implementations of BLAS and LAPACK for matrix algebra,
2. Thread support library (usually included in the compilers) for *Shared-Memory Execution*, and,
3. cURL, 0MQ and cereal for *Distributed-Memory Execution*.

Todo: Explicitly mention the versions of the required packages.

Having installed an appropriate compiler chain and CMake, we can now start with the installation procedure. First, because CMake only allows for *out-of-source* builds, we need to create a new directory (e.g., `build`) under `$HOME/polo` so that CMake can create its artifacts:

```
cd $HOME/polo
mkdir build
cd build
```

From this point, there are two options to install `polo`. The first is to check the system for the installed libraries, configure `polo` to enable those features that rely on the optional dependencies if they exist on the system, and finally install the library and configuration files under `$HOME/local`:

```
cmake -D CMAKE_INSTALL_PREFIX=$HOME/local ../src
cmake --build .
cmake --build . --target install
```

Here, we have used CMAKE_INSTALL_PREFIX to install the library and configuration files under `$HOME/local`. This is usually needed on systems where we do not have direct write access to the system directories. As a result, we need to tell CMake to also search this directory for installed libraries when we compile programs that use `polo` (we will come to this later in *Compiling, Linking and Executing*).

Note: If `-D CMAKE_INSTALL_PREFIX=$HOME/local` is dropped, and the user has the proper write access, the library and configuration files will be installed under the system directories. In the documentation, we assume that the user does *not* have administrative/system rights.

The second option to install `polo` is to use the *superbuild* feature to install all the optional dependencies and turn on all the features covered in this documentation:

```
cmake -D CMAKE_BUILD_TYPE=Release \
      -D CMAKE_INSTALL_PREFIX=$HOME/local ../
cmake --build .
cmake --build . --target install
```

Note that, this time, we use `Release` mode for CMAKE_BUILD_TYPE to install the optimized binaries of the optional dependencies.

Note: The superbuild feature requires a Fortran compiler to build BLAS and LAPACK from source.

Todo: Later, think of using Conan to package `polo` and its dependencies as prebuilt binaries.

1.1.3 Using the Prebuilt Docker Images

On some systems or architectures, installing `polo` with all the optional dependencies can be involved. To alleviate the problems in these situations, we also provide multi-architecture Docker images that contain all the optional dependencies. Using the Docker images is as simple as issuing the following:

```
docker pull pologrp/polo
docker run --tty --interactive pologrp/polo /bin/bash
```

Here, we are first pulling the latest prebuilt Docker image of `polo` for our system, and then running it in an isolated container interactively with `bash`.

Note: The rest of the documentation can be followed easily after either installing `polo` from source by using the superbuild feature or using the prebuilt Docker images.

1.2 Anatomy of a POLO Program

Programs that use `polo` are written in a single C++ file, as shown in Listing 1.1.

Note: All the code samples presented in this documentation are provided under `docs/examples` directory in the source tree. Listing captions give the relative path to the sample file. For example, `getting-started/anatomy.cpp` refers to `anatomy.cpp` file under `docs/examples/getting-started`.

Listing 1.1: getting-started/anatomy.cpp

```
1  /* include system libraries */
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  /* include polo */
7  #include <polo/polo.hpp>
8  using namespace polo;
9
10 int main(int argc, char *argv[]) {
11   /* define the problem data */
12   matrix::dmatrix<double, int> A(3, 3, {1, 0, 0, 0, 1, 0, 0, 0, 1});
13   vector<double> b{-1, 1, 1};
14   loss::data<double, int> data(A, b);
15
```

(continues on next page)

(continued from previous page)

```
  /* define the smooth loss */
  loss::logistic<double, int> loss(data);

  /* select and configure the desired solver */
  algorithm::gd<double, int> alg;
  alg.step_parameters(1.0);

  /* provide an initial vector to the solver, and solve the problem */
  const vector<double> x0{1, 1, 1};
  alg.initialize(x0);
  alg.solve(loss);

  /* print the result */
  cout << "Optimum: " << alg.getf() << '\n';
  cout << "Optimizer: [";
  for (const auto val : alg.getx())
    cout << val << ',';
  cout << "].\n";

  return 0;
}
```

As can be observed, after including the system libraries and `polo` in our C++ file, we follow the following five steps:

1. Define the problem data,
2. Define the smooth loss on the data,
3. Select and configure the desired solver,
4. Initialize the solver and run it to minimize the total loss, and,
5. Print the result.

In Listing 1.1, we first define the problem data (`data`) using the following (dense) matrix (`dmatrix`) and vector

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad b = \begin{bmatrix} -1 \\ 1 \\ 1 \end{bmatrix},$$

both of which have values of type `double` and indices of type `int`. Then, we pick a predefined loss of type `logistic` on `data`, which describes the smooth loss

$$\mathrm{f}(x) = \sum_{n=1}^{N} \log\left(1 + \exp\left(-b_n \left\langle a_n, x \right\rangle\right)\right),$$

where a_n and b_n are the n^{th} row of A and b, respectively. Currently, `polo` supports several common smooth loss functions (see *Loss Functions*). In addition, we can define custom loss functions in just a few lines of code. We will discuss this in more detail in *Defining Custom Loss Functions*. After defining the smooth loss, we select a vanilla gradient descent (`gd`) algorithm, and set its constant step size. In fact, `gd` is just an alias for a specific algorithm among a number of well-known optimization algorithms that `polo` supports from the proximal gradient family. `polo` not only allows users to configure, modify, and extend these algorithms in different ways but also supports the creation of completely new algorithms. We will describe this functionality in detail later in *Proximal Gradient Methods*. After configuring the solver, we `initialize` it with a decision vector, $x_0 = [1, 1, 1]^\top$, and `solve` the optimization problem defined on `loss`. Finally, we get the final decision vector (`getx`) that the gradient descent algorithm produces, together with the associated loss value (`getf`), and print the result.

1.3 Compiling, Linking and Executing

After writing C++ files, we need to compile the source code, and (optionally) link against different libraries to obtain the final executable. Traditionally, this compilation and linking process is merged into a single command line, in which the users provide different compile-time definitions (to configure parts of the dependencies), include directories (to search for library files), and the libraries to link against. However, this gets tedious when projects get bigger.

As we have discussed in *Installing the Library*, `polo` relies on CMake to manage its dependencies in a cross-platform compatible way. From the user's perspective, this also has the added advantage of managing the compilation and linking process much more easily.

To build code that requires `polo`, we simply create a *CMake project*. A CMake project is a directory that has a `CMakeLists.txt` file, which lists the executable files, and their dependencies in the directory. Most modern integrated development environments (IDEs) natively support CMake projects, and help users create the `CMakeLists.txt` file interactively[1]. However, we briefly mention here how to create and use CMake projects, manually, in the most basic form.

Let's assume that we have a directory, `$HOME/examples`, which contains the sample code in Listing 1.1 in a file, `anatomy.cpp`. In the same directory, we create a `CMakeLists.txt` file that has the following lines

```
cmake_minimum_required(VERSION 3.9.0)
project(polo-examples)

find_package(polo CONFIG REQUIRED)

add_executable(anatomy anatomy.cpp)
target_link_libraries(anatomy polo::polo)
```

Here, we first choose a minimum version of CMake that is required for our project, and name our project `polo-examples`. Then, we tell CMake to find the configuration files for `polo`, which is required for our project. If CMake cannot find the configuration files, the build process will stop. Otherwise, CMake will source the configuration files, and create a target library called `polo::polo`, which knows how to handle its own dependencies. Finally, for each executable we might have in the directory, we create a target executable (`add_executable`), and we link it against `polo::polo` (`target_link_libraries`). In the example, we have only one target executable, which is named `anatomy`, and it consists of only one source file, `anatomy.cpp`. Note that we do not need to define any compile-time definitions, include directories, or any of the optional dependencies of `polo` as they are already handled transitively by CMake behind the scenes.

The next step is to create a new directory (e.g., `build`) inside `$HOME/examples` for CMake to put its build artifacts. Eventually, we will end up with the following directory structure

```
cd $HOME/examples # change directory to $HOME/examples
mkdir build # make a directory named build
tree . # check the directory structure
.
├── anatomy.cpp
├── build
└── CMakeLists.txt
```

Finally, we change directory to `build`, ask CMake to build the project for us, and run the executable

```
cd build
cmake -D CMAKE_BUILD_TYPE=Release \
      -D CMAKE_PREFIX_PATH=$HOME/local ../
cmake --build .
./anatomy # could be ./anatomy.exe on Windows systems
```

[1] See, for instance, CMake Wiki for a comprehensive list of editors.

Here, we again use `Release` mode for CMAKE_BUILD_TYPE, which results in optimized binaries, and point CMAKE_PREFIX_PATH to `$HOME/local`, where we have installed `polo`, so that CMake can find and source the configuration files.

Note: If `-D CMAKE_INSTALL_PREFIX=$HOME/local` is not used in *Installing the Library*, we can drop `-D CMAKE_PREFIX_PATH=$HOME/local` when building targets that require `polo`. This is the case when `polo` is installed in system directories, or when Docker images are used.

In the end, the executable should give the following output:

```
Optimum: 0.0300018
Optimizer: [-4.57459,4.61311,4.61311,].
```

1.4 Loading Data

In *Anatomy of a POLO Program*, we have seen how to manually define problem data (cf. matrix A and vector b in Listing 1.1). This approach is appropriate for defining small data sets; however, it gets impractical for larger data sets that many machine-learning problems involve. To facilitate convenient experimentation on machine-learning problems, `polo` includes functionality[1] for reading data from common formats such as LIBSVM [2011-Chang].

Now, we briefly demonstrate how `polo` can be used to solve a logistic regression problem on data in the LIBSVM format. To this end, we revisit the example in Listing 1.1. Instead of working on the 3×3 data matrix, we `download australian_scale` file from australian data set to `data` folder under `$HOME/examples`. The data set contains 690 samples, each of which has 14 features and belongs to either of the two classes: -1 or +1. Next, we change the data-related parts of Listing 1.1 to obtain Listing 1.2 (modifications are highlighted).

Listing 1.2: getting-started/svmdata.cpp

```
1  /* include system libraries */
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  /* include polo */
7  #include <polo/polo.hpp>
8  using namespace polo;
9
10 int main(int argc, char *argv[]) {
11   /* define the problem data */
12   auto data =
13       utility::reader<double, int>::svm({"../data/australian_scale"}, 690, 14);
14
15   /* define the smooth loss */
16   loss::logistic<double, int> loss(data);
17
18   /* estimate smoothness of the loss */
19   double rowmax{0};
20   for (int row = 0; row < data.nsamples(); row++) {
21     double rowsquare{0};
22     for (const auto val : data.matrix()->getrow(row))
23       rowsquare += val * val;
24     if (rowsquare > rowmax)
```

(continues on next page)

[1] We will cover this functionality in more detail in *Data Handling*.

(continued from previous page)

```
      rowmax = rowsquare;
  }
  const double L = 0.25 * data.nsamples() * rowmax;

  /* select and configure the desired solver */
  algorithm::gd<double, int> alg;
  alg.step_parameters(2 / L);

  /* provide an initial vector to the solver, and solve the problem */
  const vector<double> x0(data.nfeatures());
  alg.initialize(x0);
  alg.solve(loss);

  /* print the result */
  cout << "Optimum: " << alg.getf() << '\n';
  cout << "Optimizer: [";
  for (const auto val : alg.getx())
    cout << val << ',';
  cout << "].\n";

  return 0;
}
```

As can be observed, we first call `svm` reader on the data file, providing the number of samples and features the file contains, and assign its output to the automatic variable `data`. Then, we try to approximate the Lipschitz smoothness of the logistic loss defined on data. This is because we are still using the vanilla gradient descent algorithm with a constant step size, and it is known that the iterates of this algorithm converge to the optimum (for convex functions) if the step size satisfies $\gamma < 2/L$, where L is the smoothness parameter [2004-Nesterov]. Because the logistic loss is defined as

$$\mathrm{f}(x) := \sum_{n=1}^{N} \mathrm{f_n}(x) = \sum_{n=1}^{N} \log\left(1 + \exp\left(-b_n \left\langle a_n, x \right\rangle\right)\right) ,$$

and each $\mathrm{f_n}(x)$ is L_n-smooth with $L_n = \|a_n\|_2^2/4$ [2014-Xiao], a computationally cheap (albeit conservative) estimate for L is

$$L = 0.25 N L_{\max} , \qquad L_{\max} = \max_n \|a_n\|_2^2 .$$

Finally, we set the step size of the algorithm, initialize it with a zero-vector of appropriate dimension, and run the algorithm. To compile Listing 1.2, we add the following lines to our previous `CMakeLists.txt`:

```
add_executable(svmdata svmdata.cpp)
target_link_libraries(svmdata polo::polo)
```

Building the executable using CMake and running the resulting program give:

```
Optimum: 229.222
Optimizer: [0.0110083,0.162899,0.0832372,0.627515,0.968077,0.328978,0.257715,1.69923,
↪0.556535,0.157199,-0.143509,0.328954,-0.358702,0.179352,].
```

1.5 Logging and Post-Processing

So far, we have seen how to define or load data, pick a loss, select and configure an algorithm, and run it to minimize a smooth loss, $\mathrm{f}(x)$. We have used `getx` and `getf` member functions of `gd` to retreive the last decision vector (i.e., *iterate*) generated by the algorithm and the smooth loss value at that iterate.

Generally, we are not only interested in the last iterate and the loss value generated by the algorithm but also the sequence of states (e.g., iterates, (partial) gradients, loss values, iteration counts, wall-clock times) the algorithm generates. To support *logging* these states while the algorithm is running, `polo` provides different *State Loggers*. Here, we briefly show how to log iteration counts, wall-clock times and the function values easily to a *comma-separated values* (csv) file.

Revisiting the example in Listing 1.2, we need to pick a proper state logger, input the logger to the algorithm, and finally save the (in-memory) logged states to a csv file. We provide the resulting code in Listing 1.3, with the necessary changes highlighted.

Listing 1.3: getting-started/logger.cpp

```
/* include system libraries */
#include <fstream>
#include <iostream>
#include <vector>
using namespace std;

/* include polo */
#include <polo/polo.hpp>
using namespace polo;

int main(int argc, char *argv[]) {
  /* define the problem data */
  auto data =
      utility::reader<double, int>::svm({"../data/australian_scale"}, 690, 14);

  /* define the smooth loss */
  loss::logistic<double, int> loss(data);

  /* estimate smoothness of the loss */
  double rowmax{0};
  for (int row = 0; row < data.nsamples(); row++) {
    double rowsquare{0};
    for (const auto val : data.matrix()->getrow(row))
      rowsquare += val * val;
    if (rowsquare > rowmax)
      rowmax = rowsquare;
  }
  const double L = 0.25 * data.nsamples() * rowmax;

  /* select and configure the desired solver */
  algorithm::gd<double, int> alg;
  alg.step_parameters(2 / L);

  /* pick a state logger */
  utility::logger::value<double, int> logger;

  /* provide an initial vector to the solver, and solve the problem */
  const vector<double> x0(data.nfeatures());
  alg.initialize(x0);
  alg.solve(loss, logger);

  /* open a csv file for writing */
  ofstream file("logger.csv");
  if (file) { /* if successfully opened for writing */
    file << "k,t,f\n";
    for (const auto &log : logger)
```

(continues on next page)

(continued from previous page)

```
      file << log << '\n';
  }

  /* print the result */
  cout << "Optimum: " << alg.getf() << '\n';
  cout << "Optimizer: [";
  for (const auto val : alg.getx())
    cout << val << ',';
  cout << "].\n";

  return 0;
}
```

First, we include the standard C++ <fstream> library to be able to open a csv file. Then, we pick a `value` logger, which logs the iteration counts, wall-clock times and the loss values generated by the algorithm, and we provide the `logger` to the `solve` method of our algorithm as the second argument. Last, for post-processing purposes, we open a csv file, named `logger.csv`, and write each `log` line by line. Note that the `value` logger, by default, outputs the iteration count, wall-clock time (in milliseconds) and the loss value in the given order, delimited by a comma.

We append the following lines to `CMakeLists.txt`

```
add_executable(logger logger.cpp)
target_link_libraries(logger polo::polo)
```

and build the project. Running the executable should give the same output as before:

```
Optimum: 229.222
Optimizer: [0.0110083,0.162899,0.0832372,0.627515,0.968077,0.328978,0.257715,1.69923,
↪0.556535,0.157199,-0.143509,0.328954,-0.358702,0.179352,].
```

However, this time, our executable has created an artifact, named `logger.csv`. We can check, for instance, the last 5 lines of the file:

```
# assuming that we are already in $HOME/examples/build
tail -n 5 logger.csv
96,5.61734,229.408
97,5.66521,229.37
98,5.70951,229.332
99,5.75266,229.295
100,5.79627,229.258
```

Moreover, we can use a plotting script such as that given in Listing 1.4 to plot the loss values with respect to iteration counts and wall-clock times.

Listing 1.4: getting-started/logger.py

```
import csv  # for reading a csv file
from matplotlib import pyplot as plt  # for plotting

k = []
t = []
f = []

with open("logger.csv") as csvfile:
    csvReader = csv.reader(csvfile, delimiter=",")
    next(csvReader)  # skip the header
```

(continues on next page)

(continued from previous page)

```
    for row in csvReader:
        k.append(int(row[0]))
        t.append(float(row[1]))
        f.append(float(row[2]))

h, w = plt.figaspect(0.5)
fig, axes = plt.subplots(1, 2, sharey=True, figsize=(h, w))

# f vs k
axes[0].plot(k, f)
axes[0].set_xlabel(r"$k$")
axes[0].set_ylabel(r"$f(\cdot)$")
axes[0].grid()

# f vs t
axes[1].plot(t, f)
axes[1].set_xlabel(r"$t$ [ms]")
axes[1].grid()

plt.tight_layout()
plt.savefig("logger.svg")
plt.savefig("logger.pdf")
```

The resulting figure should look similar to Fig. 1.1. There, we observe the loss values plotted against the iteration counts (left) and the wall-clock times (right).

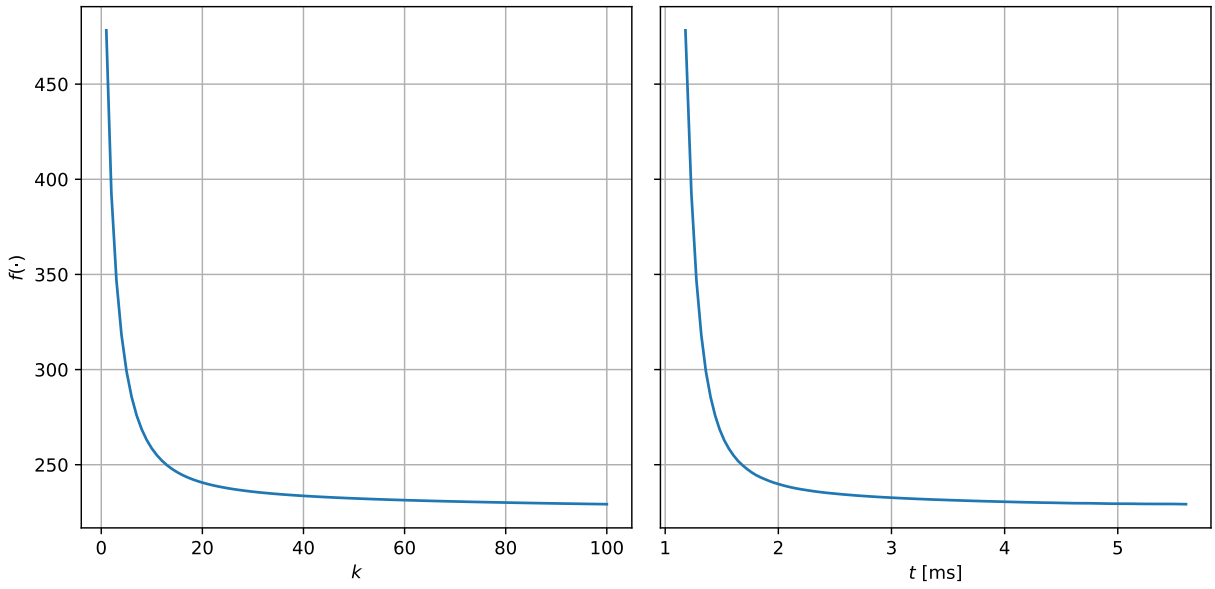

Fig. 1.1: Loss values generated by the algorithm in Listing 1.3.

Note: For this example, we have used matplotlib as the plotting library in Python. The library can be installed easily, if there exists pip on the system, by issuing `pip install --user --upgrade matplotlib`.

1.6 Terminating the Algorithm

When we check the tail of `logger.csv` file in *Logging and Post-Processing*, we realize that the iteration count, k, is at 100. It is not a coincidence; in `polo`, the default termination criterion is to have "100 iterations." Just as most parts of the algorithm, the termination criterion can also be easily changed, or even replaced by a custom *terminator*. To demonstrate this, we revisit the example in Listing 1.3, and change the default terminator to a `value` terminator, which terminates the algorithm if the change in the loss value satisfies certain conditions (see Listing 1.5).

Listing 1.5: getting-started/terminator.cpp

```
/* include system libraries */
#include <fstream>
#include <iostream>
#include <vector>
using namespace std;

/* include polo */
#include <polo/polo.hpp>
using namespace polo;

int main(int argc, char *argv[]) {
  /* define the problem data */
  auto data =
      utility::reader<double, int>::svm({"../data/australian_scale"}, 690, 14);

  /* define the smooth loss */
  loss::logistic<double, int> loss(data);

  /* estimate smoothness of the loss */
  double rowmax{0};
  for (int row = 0; row < data.nsamples(); row++) {
    double rowsquare{0};
    for (const auto val : data.matrix()->getrow(row))
      rowsquare += val * val;
    if (rowsquare > rowmax)
      rowmax = rowsquare;
  }
  const double L = 0.25 * data.nsamples() * rowmax;

  /* select and configure the desired solver */
  algorithm::gd<double, int> alg;
  alg.step_parameters(2 / L);

  /* pick a state logger */
  utility::logger::value<double, int> logger;

  /* pick a terminator */
  terminator::value<double, int> terminator(1E-3, 1E-8);

  /* provide an initial vector to the solver, and solve the problem */
  const vector<double> x0(data.nfeatures());
  alg.initialize(x0);
  alg.solve(loss, logger, terminator);

  /* open a csv file for writing */
  ofstream file("terminator.csv");
  if (file) { /* if successfully opened for writing */
```

(continues on next page)

(continued from previous page)

```
    file << "k,t,f\n";
    for (const auto &log : logger)
      file << fixed << log.getk() << ',' << log.gett() << ',' << log.getf()
           << '\n';
  }

  /* print the result */
  cout << "Optimum: " << fixed << alg.getf() << '\n';
  cout << "Optimizer: [";
  for (const auto val : alg.getx())
    cout << val << ',';
  cout << "].\n";

  return 0;
}
```

In this example, we first construct a `value` terminator with absolute and relative tolerances of `1E-3` and `1E-8`, respectively. This means that `terminator` will stop the algorithm when

$$|\mathrm{f}(x_{k-1}) - \mathrm{f}(x_k)| < \delta_{\text{abs}} = 10^{-3} \quad \text{or} \quad \left|\frac{\mathrm{f}(x_{k-1}) - \mathrm{f}(x_k)}{\mathrm{f}(x_{k-1}) + \epsilon}\right| < \delta_{\text{rel}} = 10^{-8},$$

where ϵ is, by default, the machine epsilon, and is used to prevent dividing by zero[1]. Then, we provide `terminator` to the `solve` method of our algorithm as the third argument. Finally, compared to Listing 1.3, we change how the function values are logged to `terminator.csv` and output to `cout` in Listing 1.5. In the code, we use std::fixed to print floating-point numbers in a fixed, 6-digit precision (default) format, and `getk`, `gett` and `getf` member functions of `log`[2] to get the iteration count, wall-clock time and the loss value of each logged iteration of the algorithm.

We append the following lines to `CMakeLists.txt`

```
add_executable(terminator terminator.cpp)
target_link_libraries(terminator polo::polo)
```

and build the project. Running the executable should give the output:

```
Optimum: 222.288758
Optimizer: [0.042932,0.140170,-0.352319,0.907892,1.224258,0.230179,0.558908,1.755445,
↪0.474237,0.592185,-0.117172,0.669261,-2.293000,1.374797,].
```

In this example, we realize that the "optimum" (i.e., loss value at the last iterate when the algorithm is stopped by the terminator) is lower than the one obtained in *Logging and Post-Processing*. When we check the tail of `terminator.csv`, we observe the following:

```
1307,52.607826,222.293771
1308,52.645069,222.292765
1309,52.680067,222.291760
1310,52.715353,222.290757
1311,52.751290,222.289757
```

The algorithm stops at the 1312^{th} iteration because the absolute change in the loss value becomes less than the tolerance we have asked for. Using the Python script in Listing 1.6, we obtain a figure similar to Fig. 1.2.

[1] More information on terminators is provided in *Algorithm Terminators*.
[2] More information on state loggers and their logged data is provided in *State Loggers*.

Listing 1.6: getting-started/terminator.py

```
import csv  # for reading a csv file
from matplotlib import pyplot as plt  # for plotting

k = []
t = []
f = []

with open("terminator.csv") as csvfile:
    csvReader = csv.reader(csvfile, delimiter=",")
    next(csvReader)  # skip the header
    for row in csvReader:
        k.append(int(row[0]))
        t.append(float(row[1]))
        f.append(float(row[2]))

h, w = plt.figaspect(0.5)
fig, axes = plt.subplots(1, 2, sharey=True, figsize=(h, w))

# f vs k
axes[0].plot(k, f)
axes[0].set_xlabel(r"$k$")
axes[0].set_ylabel(r"$f(\cdot)$")
axes[0].grid()

# f vs t
axes[1].plot(t, f)
axes[1].set_xlabel(r"$t$ [ms]")
axes[1].grid()

plt.tight_layout()
plt.savefig("terminator.svg")
plt.savefig("terminator.pdf")
```

In Fig. 1.2, we observe that the `value` terminator with the selected tolerances has resulted in more than 12 times more iterations (left) compared to the default `iteration` terminator with 100 iterations. Because we run both algorithms serially using one CPU, this directly translates to the wall-clock runtimes (right) of the algorithms (cf. Fig. 1.1).

1.7 Defining Custom Loss Functions

In our examples, we have thus far used the `logistic` loss on `data` that is either defined manually or read from a dataset. However, we can also define our custom loss functions, and pass them as the first argument to the `solve` member function of the algorithms. To demonstrate this, we focus on the following simple loss function:

$$\mathrm{f}(x) = \sum_{n=1}^{N} \frac{1}{n} \left(x^{(n)} - n \right)^2$$

for some $N \geq 1$. As can be observed, the loss is a convex quadratic function of the N-dimensional decision vector, and the optimizer of Problem (1) with this loss is

$$x^\star = \begin{bmatrix} 1 \\ 2 \\ \vdots \\ N \end{bmatrix} \quad \text{with} \quad \mathrm{f}\left(x^\star\right) = 0 \,.$$

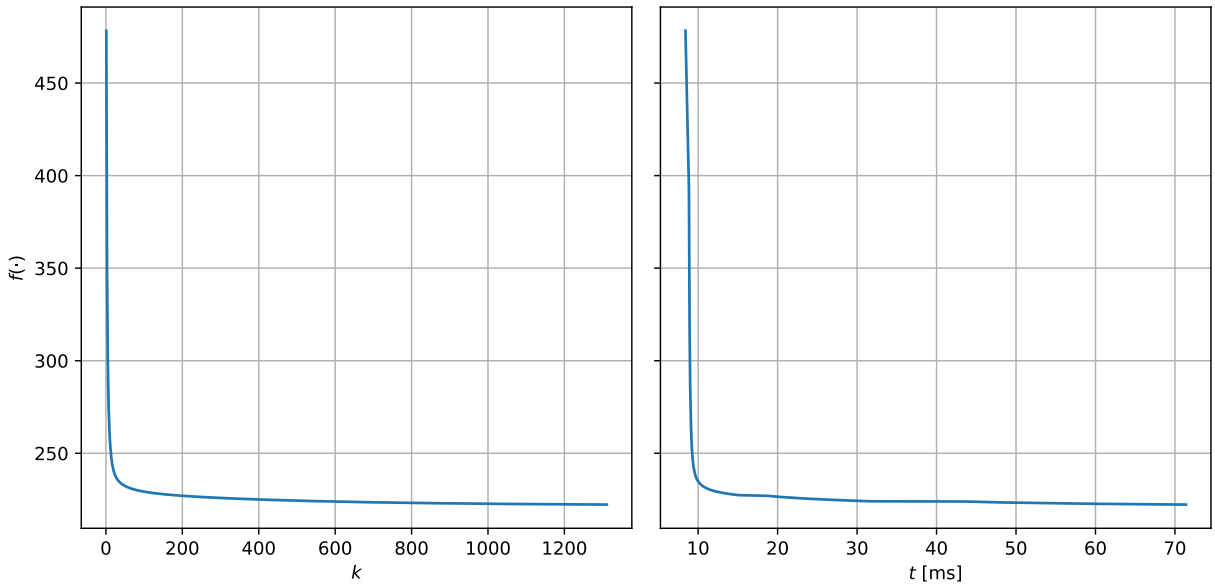

Fig. 1.2: Loss values generated by the algorithm in Listing 1.5.

Furthermore, it can easily be verified that the gradient and the Hessian of the loss are

$$\nabla \mathrm{f}(x) = \begin{bmatrix} \frac{2}{1}\left(x^{(1)} - 1\right) \\ \frac{2}{2}\left(x^{(2)} - 2\right) \\ \vdots \\ \frac{2}{N}\left(x^{(N)} - N\right) \end{bmatrix} \quad \text{and} \quad \nabla^2 \mathrm{f}(x) = \begin{bmatrix} 2 & & & \\ & 1 & & \\ & & \ddots & \\ & & & \frac{2}{N} \end{bmatrix}.$$

In `polo`, a smooth loss is *any* object that, when called with two input arguments, `x` and `g`, returns the loss value associated with `x` and writes the gradient in `g`. Because the loss objects are not allowed to modify the decision vector and because of compatibility with other (high-level) languages, the types of `x` and `g` are `const value_t *` and `value_t *`, respectively, where `value_t` is the type (e.g., `double`) of the values that the decision vector and the gradient contain. Hence, one way to define the loss mentioned above is to create a new `struct` as follows

```
struct simple_loss {
  simple_loss(const int N) : N{N} {}

  double operator()(const double *x, double *g) const {
    double loss{0};
    for (int n = 1; n <= N; n++) {
      const double residual{x[n - 1] - n};
      loss += residual * residual / n;
      g[n - 1] = 2 * residual / n;
    }
    return loss;
  }

private:
  const int N;
};
```

where we keep the data (`N`) of the loss `private`, and define the `operator()` member function appropriately. Note that, because C++ has zero-based numbering, we use `n-1` when indexing `x` and `g`.

To use a custom loss object of type `simple_loss`, we first construct `loss` with, e.g., `N = 10`. Thanks to the simple and closed-form representation of this simple loss function, we can easily verify that the loss is a *μ-strongly convex* function with $\mu = \lambda_{\text{min}} = 2/N$ and $L = \lambda_{\text{max}} = 2$, where λ_{min} and λ_{max} are the minimum and maximum eigenvalues of the Hessian, respectively. We know from [2004-Nesterov] that the vanilla gradient descent algorithm with constant step size $\gamma = \frac{2}{\mu+L}$ converges linearly to the optimum. We set the step size of the algorithm accordingly, and define a `value` terminator with absolute and relative tolerances of 10^{-8} and 10^{-13}, respectively. The resulting code is given in Listing 1.7.

Listing 1.7: getting-started/loss.cpp

```cpp
/* include system libraries */
#include <fstream>
#include <iostream>
#include <vector>
using namespace std;

/* include polo */
#include <polo/polo.hpp>
using namespace polo;

struct simple_loss {
  simple_loss(const int N) : N{N} {}

  double operator()(const double *x, double *g) const {
    double loss{0};
    for (int n = 1; n <= N; n++) {
      const double residual{x[n - 1] - n};
      loss += residual * residual / n;
      g[n - 1] = 2 * residual / n;
    }
    return loss;
  }

private:
  const int N;
};

int main(int argc, char *argv[]) {
  /* define the smooth loss */
  const int N{10};
  simple_loss loss(N);
  const double mu{2. / N};
  const double L{2.};

  /* select and configure the desired solver */
  algorithm::gd<double, int> alg;
  alg.step_parameters(2 / (mu + L));

  /* pick a state logger */
  utility::logger::value<double, int> logger;

  /* pick a terminator */
  terminator::value<double, int> terminator(1E-8, 1E-13);

  /* provide an initial vector to the solver, and solve the problem */
  const vector<double> x0(N);
  alg.initialize(x0);
```

(continues on next page)

(continued from previous page)

```
  alg.solve(loss, logger, terminator);

  /* open a csv file for writing */
  ofstream file("loss.csv");
  if (file) { /* if successfully opened for writing */
    file << "k,t,f\n";
    for (const auto &log : logger)
      file << fixed << log.getk() << ',' << log.gett() << ',' << log.getf()
           << '\n';
  }

  /* print the result */
  cout << "Optimum: " << fixed << alg.getf() << '\n';
  cout << "Optimizer: [";
  for (const auto val : alg.getx())
    cout << val << ',';
  cout << "].\n";

  return 0;
}
```

We append the following lines to `CMakeLists.txt`

```
add_executable(loss loss.cpp)
target_link_libraries(loss polo::polo)
```

and build the project. Running the executable should give the output:

```
Optimum: 0.000000
Optimizer: [1.000036,2.000000,3.000000,4.000000,5.000000,6.000000,6.999998,7.999984,8.
↪999910,9.999641,].
```

Here, we observe that the optimum value is attained (up to the fixed, 6-digit precision) by the algorithm, even though the optimizer is not. This is because the `value` terminator *only* checks the absolute and relative changes in the loss value. If we want to have a termination condition based on the changes in the decision vector instead, we can replace

```
terminator::value<double, int> terminator(1E-8, 1E-13);
```

with

```
terminator::decision<double, int> terminator(1E-8);
```

This `decision` terminator with the tolerance 10^{-8} will stop the algorithm when the following condition holds:

$$\|x_{k-1} - x_k\|_2 < \epsilon_{\text{abs}} = 10^{-8}\,.$$

Rebuilding the project and rerunning the executable should give the following:

```
Optimum: 0.000000
Optimizer: [1.000000,2.000000,3.000000,4.000000,5.000000,6.000000,7.000000,8.000000,9.
↪000000,10.000000,].
```

CHAPTER

TWO

SERIAL EXECUTION

In this chapter, we focus on solving Problem (1) sequentially on a single CPU.

2.1 Proximal Gradient Methods

One approach for solving instances of Problem (1) is to use proximal gradient methods. The basic form of the proximal gradient iteration is

$$x_{k+1} = \arg\min_{x \in \mathbb{R}^d} \left\{ \mathrm{f}(x_k) + \langle \nabla \mathrm{f}(x_k), x - x_k \rangle + \mathrm{h}(x) + \frac{1}{2\gamma_k} \|x - x_k\|_2^2 \right\}, \tag{2.1}$$

where γ_k is the step size. Thus, the next iterate, x_{k+1}, is selected to be the minimizer of the sum of the first-order approximation of the smooth loss function around the current iterate, x_k, the nonsmooth loss function, and a quadratic penalty on the deviation from the the current iterate [2017-Beck]. After some algebraic manipulations, one can rewrite Equation (2.1) in terms of the proximal operator [2017-Beck]

$$\begin{aligned} x_{k+1} &= \arg\min_{x \in \mathbb{R}^d} \left\{ \gamma_k \, \mathrm{h}(x) + \frac{1}{2} \|x - (x_k - \gamma_k \nabla \mathrm{f}(x_k))\|_2^2 \right\} \\ &:= \mathrm{prox}_{\gamma_k \, \mathrm{h}} \left(x_k - \gamma_k \nabla \mathrm{f}(x_k) \right) . \end{aligned}$$

As a result, the method can be interpreted as a two-step procedure: first, a query point is computed by modifying the current iterate in the direction of the negative gradient, and then the prox operator is applied to this query point.

Even though the proximal gradient method described in Equation (2.1) looks involved, in the sense that it requires solving an optimization problem at each iteration, the prox-mapping can actually be evaluated very efficiently for several important functions $\mathrm{h}(\cdot)$ such as, for instance, projections onto affine sets, half-spaces, boxes, and ℓ_1- and ℓ_2-norm balls. Together with its strong theoretical convergence guarantees, this makes the proximal gradient method a favorable option in many applications. However, the gradient calculation step in the vanilla proximal gradient method can be prohibitively expensive when the number of component functions (N) or the dimension of the decision vector (d) is large enough. To improve the scalability of the proximal gradient method, researchers have long tried to come up with ways of parallelizing the proximal gradient computations and more clever query points than the simple gradient step in Equation (2.1). As a result, the proximal gradient family encompasses a large variety of algorithms. We have listed some of the more influential variants in Table 2.1[1].

[1] Meanings of `boosting`, `smoothing`, `step`, `prox`, and `execution` will be clear in *Proximal Gradient Methods*.

Table 2.1: Some members of the proximal gradient methods. `s`, `cr`, `ir` and `ps` stand for serial, consistent read/write, inconsistent read/write and Parameter Server [2013-Li], respectively.

Algorithm	`boosting`	`smoothing`	`step`	`prox`	`execution`
(S)GD	×	×	γ, γ_k	×	`s`, `cr`, `ps`
IAG [2007-Blatt]	`aggregated`	×	γ	×	`s`, `cr`, `ps`
PIAG [2016-Aytekin]	`aggregated`	×	γ	✓	`s`, `cr`, `ps`
SAGA [2014-Defazio]	`saga`	×	γ	✓	`s`
Heavyball [1964-Polyak]	`classical`	×	γ	×	`s`
Nesterov [1983-Nesterov]	`nesterov`	×	γ	×	`s`
AdaGrad [2011-Duchi]	×	`adagrad`	γ	×	`s`
AdaDelta [2012-Zeiler]	×	`adadelta`	γ	×	`s`
Adam [2014-Kingma]	`classical`	`rmsprop`	γ	×	`s`
Nadam [2016-Dozat]	`nesterov`	`rmsprop`	γ	×	`s`
AdaDelay [2015-Sra]	×	×	γ_k	✓	`s`, `cr`, `ps`
HOGWILD! [2011-Recht]	×	×	γ	×	`ir`
ASAGA [2016-Leblond]	`saga`	×	γ	×	`ir`
ProxASAGA [2017-Pedregosa]	`saga`	×	γ	✓	`ir`

In the rest of the chapter, we will use different preconfigured algorithms from Table 2.1 on two different problems: *Logistic Regression* and *Least Squares Regression*.

2.2 Logistic Regression

In the *previous chapter*, we used a vanilla gradient descent (`gd`) algorithm to solve a logistic loss minimization problem. `gd` is the most basic member of the proximal gradient family.

In this section, we first revisit the logistic loss minimization problem, use different (and more advanced) members of the family, i.e., Heavyball [1964-Polyak], Nesterov [1983-Nesterov], AdaGrad [2011-Duchi] and Adam [2014-Kingma], and compare their performances. To this end, we need to change the algorithm-related parts of Listing 1.5 to obtain Listing 2.1.

Listing 2.1: serial/logistic.cpp

```
1  /* include system libraries */
2  #include <fstream>
3  #include <iostream>
4  #include <string>
5  #include <vector>
6  using namespace std;
7
8  /* include polo */
9  #include <polo/polo.hpp>
10 using namespace polo;
11
12 int main(int argc, char *argv[]) {
13   /* define the problem data */
14   auto data =
15       utility::reader<double, int>::svm({"../data/australian_scale"}, 690, 14);
16
17   /* define the smooth loss */
18   loss::logistic<double, int> loss(data);
```

(continues on next page)

(continued from previous page)

```
  /* estimate smoothness of the loss */
  double rowmax{0};
  for (int row = 0; row < data.nsamples(); row++) {
    double rowsquare{0};
    for (const auto val : data.matrix()->getrow(row))
      rowsquare += val * val;
    if (rowsquare > rowmax)
      rowmax = rowsquare;
  }
  const double L = 0.25 * data.nsamples() * rowmax;

  /* select and configure the desired solver */
#ifdef MOMENTUM
  algorithm::momentum<double, int> alg;
  alg.boosting_parameters(0.9, 1 / L);
  const string filename{"logistic-momentum.csv"};
#elif defined NESTEROV
  algorithm::nesterov<double, int> alg;
  alg.boosting_parameters(0.9, 1 / L);
  const string filename{"logistic-nesterov.csv"};
#elif defined ADAGRAD
  algorithm::adagrad<double, int> alg;
  const string filename{"logistic-adagrad.csv"};
#elif defined ADAM
  algorithm::adam<double, int> alg;
  alg.boosting_parameters(0.9, 0.1);
  alg.smoothing_parameters(0.999, 1E-8);
  alg.step_parameters(0.001);
  const string filename{"logistic-adam.csv"};
#endif

  /* pick a state logger */
  utility::logger::value<double, int> logger;

  /* pick a terminator */
  terminator::value<double, int> terminator(5E-2);

  /* provide an initial vector to the solver, and solve the problem */
  const vector<double> x0(data.nfeatures());
  alg.initialize(x0);
  alg.solve(loss, logger, terminator);

  /* open a csv file for writing */
  ofstream file(filename);
  if (file) { /* if successfully opened for writing */
    file << "k,t,f\n";
    for (const auto &log : logger)
      file << fixed << log.getk() << ',' << log.gett() << ',' << log.getf()
           << '\n';
  }

  /* print the result */
  cout << "Optimum: " << fixed << alg.getf() << '\n';
  cout << "Optimizer: [";
  for (const auto val : alg.getx())
    cout << val << ',';
```

(continues on next page)

(continued from previous page)

```
  cout << "].\n";

  return 0;
}
```

In this example, we use preprocessor directives to conditionally compile parts of the file. For instance, if the preprocessor directive `MOMENTUM` is defined, `alg` will be an instance of `momentum` (i.e., Heavyball), which updates, at each iteration k, the decision variable x_k as follows:

$$\nu_k = \mu\nu_{k-1} + \epsilon g_k$$
$$x_{k+1} = x_k - \nu_k \,,$$

where μ is the *momentum term* and usually set to 0.9 [2016-Ruder], g_k is the gradient at x_k (i.e., $g_k = \nabla \mathrm{f}(x_k)$) in the example, and ν_k is the variable that holds the exponential moving average of the gradient at iteration k. We set $\mu = 0.9$ and $\epsilon = 1/L$ by using `boosting_parameters`[1] member function of `alg`, and `filename` to `logistic-momentum.csv`, which is used for saving the logged states of the algorithm. We select and configure[1] other algorithms similarly, and guard them with the preprocessor directives. To compile Listing 2.1 for different algorithms, we add the following lines to `CMakeLists.txt`:

```
add_executable(logistic-momentum logistic.cpp)
target_compile_definitions(logistic-momentum PUBLIC MOMENTUM)
target_link_libraries(logistic-momentum polo::polo)

add_executable(logistic-nesterov logistic.cpp)
target_compile_definitions(logistic-nesterov PUBLIC NESTEROV)
target_link_libraries(logistic-nesterov polo::polo)

add_executable(logistic-adagrad logistic.cpp)
target_compile_definitions(logistic-adagrad PUBLIC ADAGRAD)
target_link_libraries(logistic-adagrad polo::polo)

add_executable(logistic-adam logistic.cpp)
target_compile_definitions(logistic-adam PUBLIC ADAM)
target_link_libraries(logistic-adam polo::polo)
```

CMake's target_compile_definitions command helps us define the appropriate preprocessor directives. Basically, using this command, we create four different executables from the same source file. Building the project, running the executables, and using the plotting script given in Listing 2.2 result in a figure similar to Fig. 2.1.

Listing 2.2: serial/logistic.py

```
import csv  # for reading a csv file
from matplotlib import pyplot as plt  # for plotting

h, w = plt.figaspect(1)
fig, axes = plt.subplots(2, 2, sharey=True, figsize=(h, w))

algorithms = ["momentum", "nesterov", "adagrad", "adam"]

for (algorithm, axis) in zip(algorithms, axes.reshape(-1)):
    k = []
    f = []

    with open(f"logistic-{algorithm}.csv") as csvfile:
```

(continues on next page)

[1] The configuration of the algorithms and the terminology will become more evident in *Proximal Gradient Methods*.

(continued from previous page)

```
        csvReader = csv.reader(csvfile, delimiter=",")
        next(csvReader)  # skip the header
        for row in csvReader:
            k.append(int(row[0]))
            f.append(float(row[2]))

    axis.plot(k, f)
    axis.set_title(algorithm)
    axis.set_xlabel(r"$k$")
    axis.set_ylabel(r"$f(\cdot)$")
    axis.grid()

plt.tight_layout()
plt.savefig("logistic.svg")
plt.savefig("logistic.pdf")
```

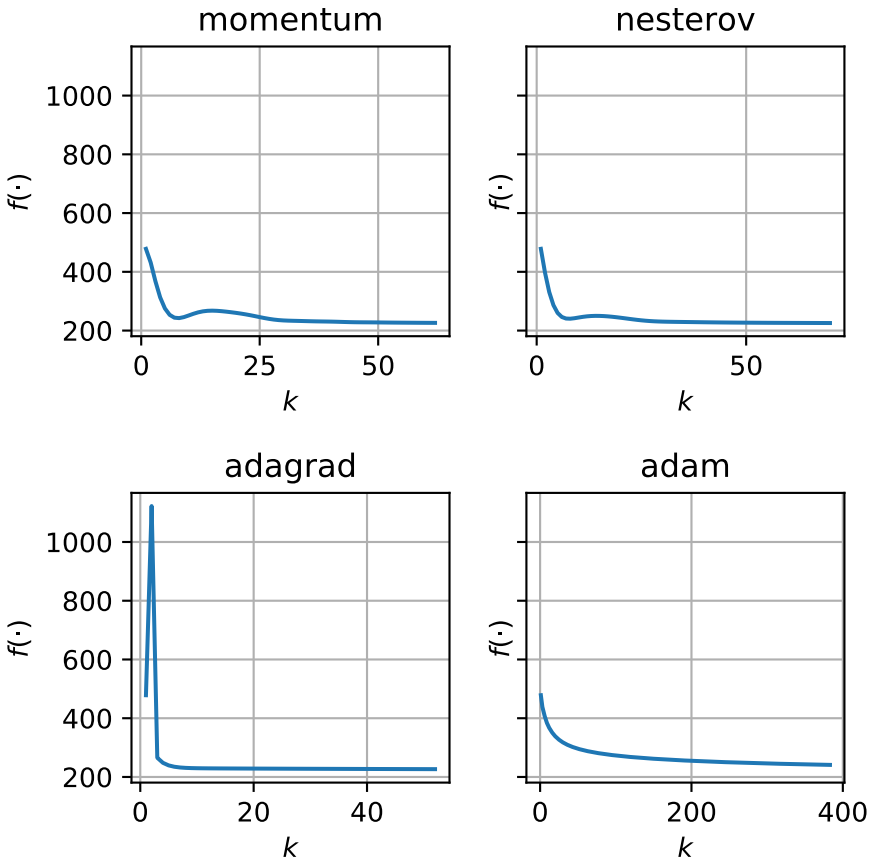

Fig. 2.1: Loss values generated by the algorithms in Listing 2.1.

In Fig. 2.1, we observe that all four algorithms converge roughly to the same loss value when a value terminator with an absolute value tolerance of `5E-2` is used. Both `momentum` and `nesterov` require the knowledge of the Lipschitz parameter (L) of the logistic loss, and they have similar convergence profiles. `adagrad` and `adam`, on the other hand, are configured *without* using the knowledge of L (cf. Listing 2.1), and they can *adapt* their learning rates to the loss function at hand. For instance, in `adagrad`, there is an increase in the loss value initially resulting from a too big initial step size, which is later accounted for as the algorithm proceeds. It is also worth noting that, in `polo`,

the default initial step size for `adagrad` is 1, and the configuration in Listing 2.1 for `adam` is the one suggested in [2014-Kingma]. Both algorithms can be tuned to have better performances than the ones obtained in this example.

Now, we add different regularizers to the problem, and repeat the same procedure to solve the corresponding regularized problems.

2.2.1 ℓ_1 Regularization

2.2.2 Elastic Net Regularization

2.3 Least Squares Regression

2.3.1 ℓ_1 Regularization

2.3.2 Elastic Net Regularization

CHAPTER

THREE

SHARED-MEMORY EXECUTION

In this chapter, we focus on solving Problem (1) using multiple CPUs in a shared-memory architecture.

3.1 Managing Shared Data

3.2 Examples

CHAPTER

FOUR

DISTRIBUTED-MEMORY EXECUTION

In this chapter, we focus on solving Problem (1) over multiple computing nodes in a distributed-memory architecture.

4.1 A Lightweight Parameter Server

4.2 Setting Up

4.2.1 Multiple Local Machines

4.2.2 Multiple Machines on AWS

4.3 An Example: Proximal Incremental Aggregated Gradient

4.4 Improving Communication

CHAPTER

FIVE

PROXIMAL GRADIENT METHODS

A careful review of the serial algorithms in the proximal gradient family reveals that, given an initial search direction, which we call a *gradient surrogate*, they differ from each other in their choices of four distinctive algorithm primitives:

1. how they combine multiple gradient surrogates to form a better search direction, a step we refer to as *boosting*,
2. how this search direction is filtered or scaled, which we call *smoothing*,
3. which *step* size policy they use, and,
4. the type of projection they do in the *prox* step.

For instance, stochastic gradient descent (SGD) algorithms use partial gradient information coming from component functions or decision vector *coordinates* as the gradient surrogate at each iteration, whereas their aggregated versions accumulate previous partial gradient information to boost the descent direction. Similarly, different momentum-based methods such as the Heavyball [1964-Polyak] and Nesterov's [1983-Nesterov] momentum accumulate the full gradient information over iterations. Algorithms such as AdaGrad [2011-Duchi] and AdaDelta [2012-Zeiler], on the other hand, use the second-moment information from the gradient surrogates and the decision vector updates to adaptively scale, i.e., smooth, the gradient surrogates. Popular algorithms such as Adam [2014-Kingma] and Nadam [2016-Dozat], available in most machine-learning libraries, incorporate both boosting and smoothing to get better update directions. Algorithms in the serial setting can also use different step size policies and projections independently of the choices above, and benefit from different *samplers* to form partial gradients as well as *encoders* to compress the gradient information. This results in the following pseudocode representation of these algorithms:

Data: Differentiable functions, $\{\mathrm{f_n}(\cdot)\}$; regularizer, $\mathrm{h}(\cdot)$.
Input: Initial decision vector, x_0.
Output: Final decision vector, x_k.
Initialize: $k \leftarrow 0$.
while $\mathrm{not\_done}(k, g, x_k, \phi(x_k))$ **do**
  $g \leftarrow \mathrm{gradient}(x_k; \mathrm{f_n}(\cdot))$; `/* full or incremental */`
  $g \leftarrow \mathrm{sample}(g)$; `/* block coordinate; optional */`
  $e \leftarrow \mathrm{encode}(g)$; `/* optional */`
  $g \leftarrow \mathrm{decode}(e)$; `/* optional */`
  $g \leftarrow \mathrm{boosting}(k, g)$; `/* optional */`
  $g \leftarrow \mathrm{smoothing}(k, g, x_k)$; `/* optional */`
  $\gamma_k \leftarrow \mathrm{step}(k, g, x_k, \phi(x_k))$;
  $x_{k+1} \leftarrow \mathrm{prox}_{\gamma_k \mathrm{h}}(x_k - \gamma_k g)$;
  $k \leftarrow k + 1$;
end
return x_k;

Most of the serial algorithms in this family either have direct counterparts in the parallel setting or can be extended to have the parallel *execution* support. However, adding parallel execution support brings yet another layer of complexity to algorithm prototyping. First, there are a variety of parallel computing environments to consider, from shared-memory and distributed-memory environments with multiple CPUs to distributed-memory heterogeneous computing environments that involve both CPUs and GPUs. Second, some of these environments, such as the shared-memory, offer different choices in how to manage race conditions. For example, some algorithms choose to use mutexes to *consistently* read from and write to the shared decision vector from different processes, whereas others prefer atomic operations to allow for *inconsistent* reads and writes. Finally, the choice in a specific computing environment might constrain choices in other algorithm primitives. For instance, if the algorithm is to run on a distributed-memory environment such as the Parameter Server [2013-Li], where only parts of the decision vector and data are stored in individual nodes, then only updates and projections that embrace data locality can be used.

In the rest of the chapter, we cover the five major policies that are used as building blocks for the proximal gradient methods.

5.1 Boosting

5.2 Smoothing

5.3 Step

5.4 Prox

Todo: Include a table of implemented prox operators.

5.5 Execution

5.5.1 Traits

CHAPTER

SIX

UTILITIES

In this chapter, we cover the `utilities` layer of `polo`.

6.1 State Loggers

6.2 Algorithm Terminators

6.3 Matrix Algebra

6.4 Data Handling

6.5 Loss Functions

6.6 Samplers

6.7 Encoders

CHAPTER

SEVEN

C-API

In this chapter, we cover the C-API of `polo` for high-level language integrations.

CHAPTER

EIGHT

POLO.JL

In this chapter, we cover `POLO.jl`, the Julia library that wraps and extends `polo` in the Julia language.

BIBLIOGRAPHY

[2004-Nesterov] Yurii E. Nesterov. *Introductory Lectures on Convex Optimization*. Springer US, 2004. DOI: 10.1007/978-1-4419-8853-9.

[2011-Chang] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM." *ACM Transactions on Intelligent Systems and Technology* 2.3 (Apr. 2011), pp. 1-27. DOI: 10.1145/1961189.1961199.

[2014-Xiao] Lin Xiao and Tong Zhang. "A Proximal Stochastic Gradient Method with Progressive Variance Reduction." *SIAM Journal on Optimization* 24.4 (Jan. 2014), pp. 2057–2075. DOI: 10.1137/140961791.

[2017-Beck] Amir Beck. *First-Order Methods in Optimization (MOS-SIAM Series on Optimization)*. Society for Industrial and Applied Mathematics (SIAM), 2017. ISBN: 978-1-611974-98-0.

[2013-Li] Mu Li et al. "Parameter Server for Distributed Machine Learning." *Big Learning Workshop, Advances in Neural Information Processing Systems 26 (NIPS).* 2013. URL: http://web.archive.org/web/20160304101521/http://www.biglearn.org/2013/files/papers/biglearning2013_submission_2.pdf

[2007-Blatt] Doron Blatt, Alfred O. Hero, and Hillel Gauchman. "A Convergent Incremental Gradient Method with a Constant Step Size." *SIAM Journal on Optimization* 18.1 (Jan. 2007), pp. 29-51. DOI: 10.1137/040615961.

[2016-Aytekin] Arda Aytekin, Hamid R. Feyzmahdavian, and Mikael Johansson. "Analysis and Implementation of an Asynchronous Optimization Algorithm for the Parameter Server." (Oct. 2016). arXiv: 1610.05507.

[2014-Defazio] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. "SAGA: A Fast Incremental Gradient Method with Support for Non-Strongly Convex Composite Objectives." *Advances in Neural Information Processing Systems 27 (NIPS).* Curran Associates, Inc., 2014, pp. 1646-1654. URL: http://papers.nips.cc/paper/5258-saga-a-fast-incremental-gradient-method-with-support-for-non-strongly-convex-composite-objectives.pdf

[2015-Sra] Suvrit Sra et al. "AdaDelay: Delay Adaptive Distributed Stochastic Convex Optimization." (Aug. 2015). arXiv: 1508.05003.

[2011-Recht] Benjamin Recht et al. "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent." *Advances in Neural Information Processing Systems 24 (NIPS).* Curran Associates, Inc., 2011, pp. 693-701. URL: http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf

[2016-Leblond] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. "Asaga: Asynchronous Parallel Saga." (June 2016). arXiv: 1606.04809.

[2017-Pedregosa] Fabian Pedregosa, Rémi Leblond, and Simon Lacoste-Julien. "Breaking the Nonsmooth Barrier: A Scalable Parallel Method for Composite Optimization." *Advances in Neural Information Processing Systems 30 (NIPS).* Curran Associates, Inc., 2017, pp. 56-65. URL: http://papers.nips.cc/paper/6611-breaking-the-nonsmooth-barrier-a-scalable-parallel-method-for-composite-optimization.pdf

[2016-Ruder] Sebastian Ruder. "An overview of gradient descent optimization algorithms." (Sep. 2016). arXiv: 1609.04747.

[1964-Polyak] Boris T. Polyak. "Some methods of speeding up the convergence of iteration methods." *USSR Computational Mathematics and Mathematical Physics* 4.5 (Jan. 1964), pp. 1-17. DOI: 10.1016/0041-5553(64)90137-5.

[1983-Nesterov] Yurii E. Nesterov. "A method for solving the convex programming problem with convergence rate $\mathcal{O}(1/k^2)$." *Soviet Mathematics Doklady* 27.2 (1983), pp. 372-376.

[2011-Duchi] John C. Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research (JMLR)* 12 (2011). pp. 2121-2159. ISSN: 1533-7928. URL: http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

[2012-Zeiler] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method." (Dec. 2012). arXiv: 1212.5701.

[2014-Kingma] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization." (Dec. 2014). arXiv: 1412.6980.

[2016-Dozat] Timothy Dozat. "Incorporating Nesterov Momentum into Adam." *ICLR Workshop*. 2016.