OptimLib

Keith O'Hara

Aug 11, 2022

GUIDE

OptimLib is a lightweight C++ library of numerical optimization methods for nonlinear functions.

Features:

- A C++11 library of local and global optimization algorithms, as well as root finding techniques.
- Derivative-free optimization using advanced, parallelized metaheuristic methods.
- Constrained optimization routines to handle simple box constraints, as well as systems of nonlinear constraints.
- For fast and efficient matrix-based computation, OptimLib supports the following templated linear algebra libraries:
  - Armadillo
  - Eigen (version >= 3.4.0)
- Automatic differentiation functionality is available through use of the Autodiff library
- OpenMP-accelerated algorithms for parallel computation.
- Straightforward linking with parallelized BLAS libraries, such as OpenBLAS.
- Available as a header-only library, or as a compiled shared library.
- Released under a permissive, non-GPL license.

Author: Keith O'Hara

License: Apache Version 2.0

CHAPTER

ONE

INSTALLATION

The library can be installed on Unix-alike systems via the standard `./configure && make` method.

See the installation page for *detailed instructions*.

CHAPTER

TWO

ALGORITHMS

A list of currently available algorithms includes:

- Broyden's Method (for root finding)
- Newton's method, BFGS, and L-BFGS
- Gradient descent: basic, momentum, Adam, AdaMax, Nadam, NadaMax, and more
- Nonlinear Conjugate Gradient
- Nelder-Mead
- Differential Evolution (DE)
- Particle Swarm Optimization (PSO)

CHAPTER

THREE

CONTENTS

3.1 Installation

OptimLib is available as a compiled shared library, or as header-only library, for Unix-alike systems only (e.g., popular Linux-based distros, as well as macOS). Note that use of this library with Windows-based systems, with or without MSVC, **is not supported**.

3.1.1 Requirements

OptimLib requires either the Armadillo or Eigen C++ linear algebra libraries. (Note that Eigen version 3.4.0 requires a C++14-compatible compiler.)

The following options should be declared **before** including the OptimLib header files.

- OpenMP functionality is enabled by default if the `_OPENMP` macro is detected (e.g., by invoking `-fopenmp` with GCC or Clang).
  - To explicitly enable OpenMP features, use:

```
#define OPTIM_USE_OPENMP
```

  - To explicitly disable OpenMP functionality, use:

```
#define OPTIM_DONT_USE_OPENMP
```

- To use OptimLib with Armadillo or Eigen:

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#define OPTIM_ENABLE_EIGEN_WRAPPERS
```

  Example:

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"
```

- To use OptimLib with RcppArmadillo or RcppEigen:

```
#define OPTIM_USE_RCPP_ARMADILLO
#define OPTIM_USE_RCPP_EIGEN
```

  Example:

```
#define OPTIM_USE_RCPP_EIGEN
#include "optim.hpp"
```

3.1.2 Installation Method 1: Shared Library

The library can be installed on Unix-alike systems via the standard `./configure && make` method.

The primary configuration options can be displayed by calling `./configure -h`, which results in:

```
$ ./configure -h

OptimLib Configuration

Main options:
-c    Code coverage build
        (default: disabled)
-d    Developmental build
        (default: disabled)
-f    Floating-point number type
        (default: double)
-g    Debugging build (optimization flags set to -O0 -g)
        (default: disabled)
-h    Print help
-i    Install path (default: current directory)
        Example: /usr/local
-l    Choice of linear algebra library
        Examples: -l arma or -l eigen
-m    Specify the BLAS and Lapack libraries to link against
        Examples: -m "-lopenblas" or -m "-framework Accelerate"
-o    Compiler optimization options
        (default: -O3 -march=native -ffp-contract=fast -flto -DARMA_NO_DEBUG)
-p    Enable OpenMP parallelization features
        (default: disabled)

Special options:
--header-only-version    Generate a header-only version of OptimLib
```

If choosing a shared library build, set (one) of the following environment variables *before* running *configure*:

```
export ARMA_INCLUDE_PATH=/path/to/armadillo
export EIGEN_INCLUDE_PATH=/path/to/eigen
```

Then, to set the install path to `/usr/local`, use Armadillo as the linear algebra library, and enable OpenMP features, we would run:

```
./configure -i "/usr/local" -l arma -p
```

Following this with the standard `make && make install` would build the library and install into `/usr/local`.

3.1.3 Installation Method 2: Header-only Library

OptimLib is also available as a header-only library (i.e., without the need to compile a shared library). Simply run `configure` with the `--header-only-version` option:

```
./configure --header-only-version
```

This will create a new directory, `header_only_version`, containing a copy of OptimLib, modified to work on an inline basis. With this header-only version, simply include the header files (`#include "optim.hpp`) and set the include path to the `head_only_version` directory (e.g.,``-I/path/to/optimlib/header_only_version``).

3.2 Examples and Tests

3.2.1 API

The OptimLib API follows a relatively simple convention, with most algorithms called using the following syntax:

```
algorithm_id(<initial/final values>, <objective function>, <objective function data>);
```

The function inputs, in order, are:

- A writable vector of initial values to define the starting point of the algorithm, where, in the event of successful completion of the algorithm, the initial values will be overwritten by the latest candidate solution vector.
- The `objective function` is a user-defined function to be minimized, or zeroed-out in the case of root finding methods.
- The final input is optional; it is any object that contains additional parameters necessary to evaluate the objective function.

For example, the BFGS algorithm is called using

```
bfgs(ColVec_t& init_out_vals, std::function<double (const ColVec_t& vals_inp, ColVec_t*␣
↪grad_out, void* opt_data)> opt_objfn, void* opt_data);
```

3.2.2 Example

The code below uses Differential Evolution to search for the minimum of the *Ackley function*.

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

// Ackley function

double ackley_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    const double x = vals_inp(0);
    const double y = vals_inp(1);
    const double pi = arma::datum::pi;

    double obj_val = -20*std::exp( -0.2*std::sqrt(0.5*(x*x + y*y)) ) - std::exp( 0.
↪5*(std::cos(2*pi*x) + std::cos(2*pi*y)) ) + 22.718282L;

    //

    return obj_val;
}

int main()
{
    // initial values:
    arma::vec x = arma::ones(2,1) + 1.0; // (2,2)

    //

    std::chrono::time_point<std::chrono::system_clock> start = std::chrono::system_
↪clock::now();

    bool success = optim::de(x,ackley_fn,nullptr);

    std::chrono::time_point<std::chrono::system_clock> end = std::chrono::system_
↪clock::now();
    std::chrono::duration<double> elapsed_seconds = end-start;

    if (success) {
        std::cout << "de: Ackley test completed successfully.\n"
                  << "elapsed time: " << elapsed_seconds.count() << "s\n";
    } else {
        std::cout << "de: Ackley test completed unsuccessfully." << std::endl;
    }

    arma::cout << "\nde: solution to Ackley test:\n" << x << arma::endl;

    return 0;
}
```

On x86-based computers, this example can be compiled using:

```
g++ -Wall -std=c++11 -O3 -march=native -ffp-contract=fast -I/path/to/armadillo -I/path/
↪to/optim/include optim_de_ex.cpp -o optim_de_ex.out -L/path/to/optim/lib -loptim
```

3.2.3 Test suite

You can build the test suite as follows:

```
# compile tests
cd ./tests
./setup
cd ./unconstrained
./configure -l arma
make
./bfgs.test
```

3.3 Optimization Settings

3.3.1 Main

An object of type `algo_settings_t` can be used to control the behavior of the optimization routines. Each algorithm page details the relevant parameters for that methods, but we list the full settings here for completeness.

```
struct algo_settings_t
{
    // RNG seeding

    size_t rng_seed_value = std::random_device{}();
```

(continues on next page)

(continued from previous page)

```
// print and convergence options

int print_level = 0;
int conv_failure_switch = 0;

// error tolerance and maxiumum iterations

size_t iter_max = 2000;

fp_t grad_err_tol  = 1E-08;
fp_t rel_sol_change_tol  = 1E-14;
fp_t rel_objfn_change_tol = 1E-08;

// bounds

bool vals_bound = false;

ColVec_t lower_bounds;
ColVec_t upper_bounds;

// values returned upon successful completion

fp_t opt_fn_value;      // will be returned by the optimization algorithm
ColVec_t opt_root_fn_values; // will be returned by the root-finding method

size_t opt_iter;
fp_t opt_error_value;

// algorithm-specific parameters

// BFGS
bfgs_settings_t bfgs_settings;

// CG
cg_settings_t cg_settings;

// DE
de_settings_t de_settings;

// GD
gd_settings_t gd_settings;

// L-BFGS
lbfgs_settings_t lbfgs_settings;

// Nelder-Mead
nm_settings_t nm_settings;

// PSO
pso_settings_t pso_settings;
```

(continues on next page)

(continued from previous page)

```
    // SUMT
    sumt_settings_t sumt_settings;

    // Broyden
    broyden_settings_t broyden_settings;
};
```

Description:

- `rng_seed_value` seed value used for random number generators.
- `print_level` sets the level of detail for printing updates on optimization algorithm progress.
- `conv_failure_switch` policy regarding what to return when an error is encountered.
- `iter_max` maximum number of iterations.
- `grad_err_tol` tolerance value controlling gradient-based convergence.
- `rel_sol_change_tol` tolerance value controlling convergence based on the relative change in optimal input values.
- `rel_objfn_change_tol` tolerance value controlling convergence based on the relative change in objective function.
- `vals_bound` whether the search space of the algorithm is bounded.
- `lower_bounds` defines the lower bounds of the search space.
- `upper_bounds` defines the upper bounds of the search space.
- `opt_fn_value` value of the objection function when evaluated at the optimal input values.
- `opt_root_fn_values` values of the root functions when evaluated at the optimal input values.
- `opt_iter` number of iterations before convergence was declared
- `opt_error_value` error value at the optimum input values

Algorithm-specific data structures are listed in the next section.

3.3.2 By Algorithm

BFGS

```
struct bfgs_settings_t
{
    fp_t wolfe_cons_1 = 1E-03; // line search tuning parameter
    fp_t wolfe_cons_2 = 0.90;  // line search tuning parameter
};
```

Conjugate Gradient

```
struct cg_settings_t
{
    bool use_rel_sol_change_crit = false;
    int method = 2;
    fp_t restart_threshold = 0.1;

    fp_t wolfe_cons_1 = 1E-03; // line search tuning parameter
    fp_t wolfe_cons_2 = 0.10;  // line search tuning parameter
};
```

Gradient Descent

```
struct gd_settings_t
{
    int method = 0;

    // step size, or 'the learning rate'
    fp_t par_step_size = 0.1;

    // decay
    bool step_decay = false;

    uint_t step_decay_periods = 10;
    fp_t step_decay_val = 0.5;

    // momentum parameter
    fp_t par_momentum = 0.9;

    // Ada parameters
    fp_t par_ada_norm_term = 1.0e-08;

    fp_t par_ada_rho = 0.9;

    bool ada_max = false;

    // Adam parameters
    fp_t par_adam_beta_1 = 0.9;
    fp_t par_adam_beta_2 = 0.999;

    // gradient clipping settings
    bool clip_grad = false;

    bool clip_max_norm = false;
    bool clip_min_norm = false;
    int clip_norm_type = 2;
    fp_t clip_norm_bound = 5.0;
};
```

L-BFGS

```
struct lbfgs_settings_t
{
    size_t par_M = 10;

    fp_t wolfe_cons_1 = 1E-03; // line search tuning parameter
    fp_t wolfe_cons_2 = 0.90;  // line search tuning parameter
};
```

Nelder-Mead

```
struct nm_settings_t
{
    bool adaptive_pars = true;

    fp_t par_alpha = 1.0; // reflection parameter
    fp_t par_beta  = 0.5; // contraction parameter
    fp_t par_gamma = 2.0; // expansion parameter
    fp_t par_delta = 0.5; // shrinkage parameter

    bool custom_initial_simplex = false;
    Mat_t initial_simplex_points;
};
```

Differential Evolution

```
struct de_settings_t
{
    size_t n_pop = 200;
    size_t n_pop_best = 6;
    size_t n_gen = 1000;

    int omp_n_threads = -1; // numbers of threads to use

    int mutation_method = 1; // 1 = rand; 2 = best

    size_t check_freq = (size_t)-1;

    fp_t par_F = 0.8;
    fp_t par_CR = 0.9;

    // DE-PRMM specific

    int pmax = 4;
    size_t max_fn_eval = 100000;

    fp_t par_F_l = 0.1;
    fp_t par_F_u = 1.0;
```

(continues on next page)

(continued from previous page)

```
    fp_t par_tau_F  = 0.1;
    fp_t par_tau_CR = 0.1;

    fp_t par_d_eps = 0.5;

    // initial vals

    ColVec_t initial_lb; // this will default to -0.5
    ColVec_t initial_ub; // this will default to  0.5

    //

    bool return_population_mat = false;
    Mat_t population_mat; // n_pop x n_vals
};
```

Particle Swarm Optimization

```
struct pso_settings_t
{
    bool center_particle = true;

    size_t n_pop = 100;
    size_t n_gen = 1000;

    int omp_n_threads = -1; // numbers of threads to use

    int inertia_method = 1; // 1 for linear decreasing between w_min and w_max; 2 for␣
↪dampening

    size_t check_freq = (size_t)-1;

    fp_t par_initial_w = 1.0;
    fp_t par_w_damp = 0.99;

    fp_t par_w_min = 0.10;
    fp_t par_w_max = 0.99;

    int velocity_method = 1; // 1 for fixed; 2 for linear

    fp_t par_c_cog = 2.0;
    fp_t par_c_soc = 2.0;

    fp_t par_initial_c_cog = 2.5;
    fp_t par_final_c_cog   = 0.5;
    fp_t par_initial_c_soc = 0.5;
    fp_t par_final_c_soc   = 2.5;

    ColVec_t initial_lb; // this will default to -0.5
    ColVec_t initial_ub; // this will default to  0.5
```

(continues on next page)

(continued from previous page)

```
    //

    bool return_position_mat = false;
    Mat_t position_mat; // n_pop x n_vals
};
```

SUMT

```
struct sumt_settings_t
{
    fp_t par_eta = 10.0;
};
```

Broyden

```
struct broyden_settings_t
{
    fp_t par_rho = 0.9;
    fp_t par_sigma_1 = 0.001;
    fp_t par_sigma_2 = 0.001;
};
```

3.4 Automatic Differentiation

Gradient-based optimization methods in OptimLib (such as BFGS and gradient descent) require a user-defined function that returns a gradient vector at each function evaluation. While this is best achieved by knowing the gradient in closed form, OptimLib also provides **experimental support** for automatic differentiation via the autodiff library.

Requirements: an Eigen-based build of OptimLib, a copy of the `autodiff` header files, and a C++17 compatible compiler.

3.4.1 Example

The example below uses forward-mode automatic differentiation to compute the gradient of the Sphere function, and the BFGS algorithm to find the input values that minimize the autodiff-enabled function.

```
/*
* Forward-mode autodiff test with Sphere function
*/

#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

#include <autodiff/forward/real.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <autodiff/forward/real/eigen.hpp>

//

autodiff::real
opt_fnd(const autodiff::ArrayXreal& x)
{
    return x.cwiseProduct(x).sum();
}

double
opt_fn(const Eigen::VectorXd& x, Eigen::VectorXd* grad_out, void* opt_data)
{
    autodiff::real u;
    autodiff::ArrayXreal xd = x.eval();

    if (grad_out) {
        Eigen::VectorXd grad_tmp = autodiff::gradient(opt_fnd, autodiff::wrt(xd),␣
↪autodiff::at(xd), u);

        *grad_out = grad_tmp;
    } else {
        u = opt_fnd(xd);
    }

    return u.val();
}

int main()
{
    Eigen::VectorXd x(5);
    x << 1, 2, 3, 4, 5;

    bool success = optim::bfgs(x, opt_fn, nullptr);

    if (success) {
        std::cout << "bfgs: forward-mode autodiff test completed successfully.\n" <<␣
↪std::endl;
    } else {
        std::cout << "bfgs: forward-mode autodiff test completed unsuccessfully.\n" <<␣
↪std::endl;
    }

    std::cout << "solution: x = \n" << x << std::endl;

    return 0;
}
```

This example can be compiled using:

```
g++ -Wall -std=c++17 -O3 -march=native -ffp-contract=fast -I/path/to/eigen -I/path/to/
↪autodiff -I/path/to/optim/include optim_autodiff_ex.cpp -o optim_autodiff_ex.out -L/
```

(continues on next page)

(continued from previous page)

```
↪path/to/optim/lib -loptim
```

See the `examples/autodiff` directory for an example using reverse-mode automatic differentiation.

3.5 Convex Optimization

3.5.1 BFGS

Table of contents

Algorithm Description

The Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm is a quasi-Newton optimization method that can be used solve to optimization problems of the form

$$\min_{x \in X} f(x)$$

where $f : \mathbb{R}^d \to \mathbb{R}$ is convex and twice differentiable. The BFGS algorithm requires that the gradient of f be known and forms an approximation to the Hessian.

The updating rule for BFGS is described below. Let $x^{(i)}$ denote the candidate solution vector at stage i of the algorithm.

1. Compute the descent direction using:

$$d^{(i)} = -W^{(i)}[\nabla_x f(x^{(i)})]$$

   where W is an approximation to the inverse Hessian matrix (the calculation of which is described in step 4).

2. Compute the optimal step size using line search:

$$\alpha^{(i)} = \arg\min_{\alpha} f(x^{(i)} + \alpha \times d^{(i)})$$

3. Update the candidate solution vector using:

$$x^{(i+1)} = x^{(i)} + \alpha^{(i)} \times d^{(i)}$$

4. Update the approximation to the inverse Hessian matrix W using the updating rule:

$$W^{(i+1)} = \left(I - P^{(i+1)}\right) W^{(i)} \left(I - P^{(i+1)}\right)^{\top} + \frac{1}{[y^{(i+1)}]^{\top} s^{(i+1)}} s^{(i+1)} [s^{(i+1)}]^{\top}$$

where

$$P^{(i)} = \frac{1}{[y^{(i)}]^{\top} s^{(i)}} s^{(i)} [y^{(i)}]^{\top}$$
$$s^{(i)} := x^{(i)} - x^{(i-1)}$$
$$y^{(i)} := \nabla_x f(x^{(i)}) - \nabla_x f(x^{(i-1)})$$

The algorithm stops when at least one of the following conditions are met:

1. the norm of the gradient vector, $\|\nabla f\|$, is less than `grad_err_tol`;
2. the relative change between $x^{(i+1)}$ and $x^{(i)}$ is less than `rel_sol_change_tol`;
3. the total number of iterations exceeds `iter_max`.

Function Declarations

bool **bfgs**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data)

The Broyden–Fletcher–Goldfarb–Shanno (BFGS) Quasi-Newton Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **bfgs**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

The Broyden–Fletcher–Goldfarb–Shanno (BFGS) Quasi-Newton Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:

  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **`opt_data`** – additional data passed to the user-provided function.
- **`settings`** – parameters controlling the optimization routine.

Returns
a boolean value indicating successful completion of the optimization algorithm.

Optimization Control Parameters

The basic control parameters are:

- `fp_t grad_err_tol`: the error tolerance value controlling how small the L_2 norm of the gradient vector $\|\nabla f\|$ should be before 'convergence' is declared.
- `fp_t rel_sol_change_tol`: the error tolerance value controlling how small the proportional change in the solution vector should be before 'convergence' is declared.

  The relative change is computed using:

$$\left\| \frac{x^{(i)} - x^{(i-1)}}{|x^{(i-1)}| + \epsilon} \right\|_1$$

  where ϵ is a small number added for numerical stability.
- `size_t iter_max`: the maximum number of iterations/updates before the algorithm exits.
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

Additional settings:

- `fp_t bfgs_settings.wolfe_cons_1`: Line search tuning parameter that controls the tolerance on the Armijo sufficient decrease condition.
  - Default value: `1E-03`.
- `fp_t bfgs_settings.wolfe_cons_2`: Line search tuning parameter that controls the tolerance on the curvature condition.
  - Default value: `0.90`.
- `int print_level`: Set the level of detail for printing updates on optimization progress.
  - Level `0`: Nothing (default).
  - Level `1`: Print the current iteration count and error values.
  - Level `2`: Level 1 plus the current candidate solution values, $x^{(i+1)}$.
  - Level `3`: Level 2 plus the direction vector, $d^{(i)}$, and the gradient vector, $\nabla_x f(x^{(i+1)})$.

- Level 4: Level 3 plus the components used to update the approximate inverse Hessian matrix: $s^{(i+1)}$, $y^{(i+1)}$, and $W^{(i+1)}$.

Examples

Sphere Function

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
sphere_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double obj_val = arma::dot(vals_inp,vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
    const int test_dim = 5;

    arma::vec x = arma::ones(test_dim,1); // initial values (1,1,...,1)

    bool success = optim::bfgs(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "bfgs: sphere test completed successfully." << "\n";
    } else {
        std::cout << "bfgs: sphere test completed unsuccessfully." << "\n";
    }

    arma::cout << "bfgs: solution to sphere test:\n" << x << arma::endl;

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"
```

(continues on next page)

(continued from previous page)

```
inline
double
sphere_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double obj_val = vals_inp.dot(vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
    const int test_dim = 5;

    Eigen::VectorXd x = Eigen::VectorXd::Ones(test_dim); // initial values (1,1,...,1)

    bool success = optim::bfgs(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "bfgs: sphere test completed successfully." << "\n";
    } else {
        std::cout << "bfgs: sphere test completed unsuccessfully." << "\n";
    }

    std::cout << "bfgs: solution to sphere test:\n" << x << std::endl;

    return 0;
}
```

Booth's Function

Code to run this example is given below.

Armadillo Code (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
booth_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    double obj_val = std::pow(x_1 + 2*x_2 - 7.0,2) + std::pow(2*x_1 + x_2 - 5.0,2);
```

(continues on next page)

(continued from previous page)

```
    if (grad_out) {
        (*grad_out)(0) = 10*x_1 + 8*x_2   2*(- 7.0) + 4*(x_2 - 5.0);
        (*grad_out)(1) = 2*(x_1 + 2*x_2 - 7.0)*2 + 2*(2*x_1 + x_2 - 5.0);
    }

    return obj_val;
}

int main()
{
    arma::vec x_2 = arma::zeros(2,1); // initial values (0,0)

    bool success_2 = optim::bfgs(x, booth_fn, nullptr);

    if (success_2) {
        std::cout << "bfgs: Booth test completed successfully." << "\n";
    } else {
        std::cout << "bfgs: Booth test completed unsuccessfully." << "\n";
    }

    arma::cout << "bfgs: solution to Booth test:\n" << x_2 << arma::endl;

    return 0;
}
```

Eigen Code (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
double
booth_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    double obj_val = std::pow(x_1 + 2*x_2 - 7.0,2) + std::pow(2*x_1 + x_2 - 5.0,2);

    if (grad_out) {
        (*grad_out)(0) = 2*(x_1 + 2*x_2 - 7.0) + 2*(2*x_1 + x_2 - 5.0)*2;
        (*grad_out)(1) = 2*(x_1 + 2*x_2 - 7.0)*2 + 2*(2*x_1 + x_2 - 5.0);
    }

    return obj_val;
}

int main()
{
    Eigen::VectorXd x = Eigen::VectorXd::Zero(2); // initial values (0,0)

    bool success_2 = optim::bfgs(x, booth_fn, nullptr);
```

(continues on next page)

(continued from previous page)

```
    if (success_2) {
        std::cout << "bfgs: Booth test completed successfully." << "\n";
    } else {
        std::cout << "bfgs: Booth test completed unsuccessfully." << "\n";
    }

    std::cout << "bfgs: solution to Booth test:\n" << x_2 << std::endl;

    return 0;
}
```

bfgs	BFGS
bfgs	BFGS

3.5.2 Limited Memory BFGS

Table of contents

Algorithm Description

The limited memory variant of the Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm is a quasi-Newton optimization method that can be used solve to optimization problems of the form

$$\min_{x \in X} f(x)$$

where f is convex and twice differentiable. The BFGS algorithm requires that the gradient of f be known and forms an approximation to the Hessian.

The updating rule for L-BFGS is described below. Let $x^{(i)}$ denote the candidate solution vector at stage i of the algorithm.

1. Compute the descent direction using a two loop recursion. Let

$$s^{(i)} := x^{(i+1)} - x^{(i)}$$
$$y^{(i)} := \nabla_x f(x^{(i+1)}) - \nabla_x f(x^{(i)})$$

   Then

a. Set

$$q = \nabla_x f(x^{(i)})$$

b. For $k = i-1, \ldots, i-M$ do:

$$\rho_k := \frac{1}{[y^{(k)}]^\top s^{(k)}}$$
$$\gamma_k := \rho_k \times [s^{(k)}]^\top q$$
$$q = q - \gamma_k \times y^{(k)}$$

where M, the history length, is set via `par_M`.

c. Set

$$r = q \times \frac{[s^{(i)}]^\top y^{(i)}}{[y^{(i)}]^\top y^{(i)}}$$

d. For $k = i-M, \ldots, i-1$ do:

$$\beta := \rho_k \times [y^{(k)}]^\top r$$
$$r = r + s^{(k)}(\gamma_k - \beta)$$

e. Set

$$d^{(i)} = -r$$

2. Compute the optimal step size using line search:

$$\alpha^{(i)} = \arg\min_\alpha f(x^{(i)} + \alpha \times d^{(i)})$$

3. Update the candidate solution vector using:

$$x^{(i+1)} = x^{(i)} + \alpha^{(i)} \times d^{(i)}$$

The algorithm stops when at least one of the following conditions are met:

1. the norm of the gradient vector, $\|\nabla f\|$, is less than `grad_err_tol`;
2. the relative change between $x^{(i+1)}$ and $x^{(i)}$ is less than `rel_sol_change_tol`;
3. the total number of iterations exceeds `iter_max`.

Function Declarations

bool **lbfgs**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data)

The Limited Memory Variant of the BFGS Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.

- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.

Returns
a boolean value indicating successful completion of the optimization algorithm.

bool **lbfgs**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

The Limited Memory Variant of the BFGS Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns
a boolean value indicating successful completion of the optimization algorithm.

Optimization Control Parameters

The basic control parameters are:

- `fp_t grad_err_tol`: the error tolerance value controlling how small the L_2 norm of the gradient vector $\|\nabla f\|$ should be before 'convergence' is declared.
- `fp_t rel_sol_change_tol`: the error tolerance value controlling how small the proportional change in the solution vector should be before 'convergence' is declared.

  The relative change is computed using:

$$\left\| \frac{x^{(i)} - x^{(i-1)}}{|x^{(i-1)}| + \epsilon} \right\|_1$$

  where ϵ is a small number added for numerical stability.
- `size_t iter_max`: the maximum number of iterations/updates before the algorithm exits.
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.

  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

Additional settings:

- `size_t lbfgs_settings.par_M`: The number of past gradient vectors to use when forming the approximate Hessian matrix.
  - Default value: `10`.
- `fp_t lbfgs_settings.wolfe_cons_1`: Line search tuning parameter that controls the tolerance on the Armijo sufficient decrease condition.
  - Default value: `1E-03`.
- `fp_t lbfgs_settings.wolfe_cons_2`: Line search tuning parameter that controls the tolerance on the curvature condition.
  - Default value: `0.90`.
- `int print_level`: Set the level of detail for printing updates on optimization progress.
  - Level `0`: Nothing (default).
  - Level `1`: Print the current iteration count and error values.
  - Level `2`: Level 1 plus the current candidate solution values, $x^{(i+1)}$.
  - Level `3`: Level 2 plus the direction vector, $d^{(i)}$, and the gradient vector, $\nabla_x f(x^{(i+1)})$.
  - Level `4`: Level 3 plus print components used to update the approximate inverse Hessian matrix: s and y.

Examples

Sphere Function

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
sphere_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double obj_val = arma::dot(vals_inp,vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
```

(continues on next page)

(continued from previous page)

```
    const int test_dim = 5;

    arma::vec x = arma::ones(test_dim,1); // initial values (1,1,...,1)

    bool success = optim::lbfgs(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "bfgs: sphere test completed successfully." << "\n";
    } else {
        std::cout << "bfgs: sphere test completed unsuccessfully." << "\n";
    }

    arma::cout << "bfgs: solution to sphere test:\n" << x << arma::endl;

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
double
sphere_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double obj_val = vals_inp.dot(vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
    const int test_dim = 5;

    Eigen::VectorXd x = Eigen::VectorXd::Ones(test_dim); // initial values (1,1,...,1)

    bool success = optim::lbfgs(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "bfgs: sphere test completed successfully." << "\n";
    } else {
        std::cout << "bfgs: sphere test completed unsuccessfully." << "\n";
    }

    std::cout << "bfgs: solution to sphere test:\n" << x << std::endl;

    return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

Booth's Function

Code to run this example is given below.

Armadillo Code (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
booth_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    double obj_val = std::pow(x_1 + 2*x_2 - 7.0,2) + std::pow(2*x_1 + x_2 - 5.0,2);

    if (grad_out) {
        (*grad_out)(0) = 10*x_1 + 8*x_2   2*(- 7.0) + 4*(x_2 - 5.0);
        (*grad_out)(1) = 2*(x_1 + 2*x_2 - 7.0)*2 + 2*(2*x_1 + x_2 - 5.0);
    }

    return obj_val;
}

int main()
{
    arma::vec x_2 = arma::zeros(2,1); // initial values (0,0)

    bool success_2 = optim::lbfgs(x, booth_fn, nullptr);

    if (success_2) {
        std::cout << "bfgs: Booth test completed successfully." << "\n";
    } else {
        std::cout << "bfgs: Booth test completed unsuccessfully." << "\n";
    }

    arma::cout << "bfgs: solution to Booth test:\n" << x_2 << arma::endl;

    return 0;
}
```

Eigen Code (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"
```

(continues on next page)

(continued from previous page)

```
inline
double
booth_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    double obj_val = std::pow(x_1 + 2*x_2 - 7.0,2) + std::pow(2*x_1 + x_2 - 5.0,2);

    if (grad_out) {
        (*grad_out)(0) = 2*(x_1 + 2*x_2 - 7.0) + 2*(2*x_1 + x_2 - 5.0)*2;
        (*grad_out)(1) = 2*(x_1 + 2*x_2 - 7.0)*2 + 2*(2*x_1 + x_2 - 5.0);
    }

    return obj_val;
}

int main()
{
    Eigen::VectorXd x = Eigen::VectorXd::Zero(test_dim); // initial values (0,0)

    bool success_2 = optim::lbfgs(x, booth_fn, nullptr);

    if (success_2) {
        std::cout << "bfgs: Booth test completed successfully." << "\n";
    } else {
        std::cout << "bfgs: Booth test completed unsuccessfully." << "\n";
    }

    std::cout << "bfgs: solution to Booth test:\n" << x_2 << std::endl;

    return 0;
}
```

lbfgs	Limited Memory variant of BFGS
lbfgs	Limited Memory variant of BFGS

3.5.3 Conjugate Gradient

Table of contents

- *Sphere Function*
- *Booth's Function*

Algorithm Description

The Nonlinear Conjugate Gradient (CG) algorithm is used solve to optimization problems of the form

$$\min_{x \in X} f(x)$$

where f is convex and (at least) once differentiable. The algorithm requires the gradient function to be known.

The updating rule for CG is described below. Let $x^{(i)}$ denote the function input values at stage i of the algorithm.

1. Compute the descent direction using:

$$d^{(i)} = -[\nabla_x f(x^{(i)})] + \beta^{(i)} d^{(i-1)}$$

2. Determine if $\beta^{(i)}$ should be reset (to zero), which occurs when

$$\frac{|[\nabla f(x^{(i)})]^\top [\nabla f(x^{(i-1)})]|}{[\nabla f(x^{(i)})]^\top [\nabla f(x^{(i)})]} > \nu$$

where ν is set via `cg_settings.restart_threshold`.

3. Compute the optimal step size using line search:

$$\alpha^{(i)} = \arg\min_{\alpha} f(x^{(i)} + \alpha \times d^{(i)})$$

4. Update the candidate solution vector using:

$$x^{(i+1)} = x^{(i)} + \alpha^{(i)} \times d^{(i)}$$

The algorithm stops when at least one of the following conditions are met:

1. the norm of the gradient vector, $\|\nabla f\|$, is less than `grad_err_tol`;
2. the relative change between $x^{(i+1)}$ and $x^{(i)}$ is less than `rel_sol_change_tol`;
3. the total number of iterations exceeds `iter_max`.

Updating Rules

- `cg_settings.method = 1` Fletcher–Reeves (FR):

$$\beta_{\text{FR}} = \frac{[\nabla_x f(x^{(i)})]^\top [\nabla_x f(x^{(i)})]}{[\nabla_x f(x^{(i-1)})]^\top [\nabla_x f(x^{(i-1)})]}$$

- `cg_settings.method = 2` Polak-Ribiere (PR):

$$\beta_{\text{PR}} = \frac{[\nabla_x f(x^{(i)})]^\top [\nabla_x f(x^{(i)})]}{[\nabla_x f(x^{(i-1)})]^\top [\nabla_x f(x^{(i-1)})]}$$

- `cg_settings.method = 3` FR-PR Hybrid:

$$\beta = \begin{cases} -\beta_{\text{FR}} & \text{if } \beta_{\text{PR}} < -\beta_{\text{FR}} \\ \beta_{\text{PR}} & \text{if } |\beta_{\text{PR}}| \leq \beta_{\text{FR}} \\ \beta_{\text{FR}} & \text{if } \beta_{\text{PR}} > \beta_{\text{FR}} \end{cases}$$

- `cg_settings.method = 4` Hestenes-Stiefel:

$$\beta_{\text{HS}} = \frac{[\nabla_x f(x^{(i)})] \cdot ([\nabla_x f(x^{(i)})] - [\nabla_x f(x^{(i-1)})])}{([\nabla_x f(x^{(i)})] - [\nabla_x f(x^{(i-1)})]) \cdot d^{(i)}}$$

- `cg_settings.method = 5` Dai-Yuan:

$$\beta_{\text{DY}} = \frac{[\nabla_x f(x^{(i)})] \cdot [\nabla_x f(x^{(i)})]}{([\nabla_x f(x^{(i)})] - [\nabla_x f(x^{(i-1)})]) \cdot d^{(i)}}$$

- `cg_settings.method = 6` Hager-Zhang:

$$\beta_{\text{HZ}} = \left(y^{(i)} - 2 \times \frac{[y^{(i)}] \cdot y^{(i)}}{y^{(i)} \cdot d^{(i)}} \times d^{(i)} \right) \cdot \frac{[\nabla_x f(x^{(i)})]}{y^{(i)} \cdot d^{(i)}}$$
$$y^{(i)} := [\nabla_x f(x^{(i)})] - [\nabla_x f(x^{(i-1)})]$$

Finally, we set:

$$\beta^{(i)} = \max\{0, \beta_*\}$$

where β_* is the update method chosen.

Function Declarations

bool **cg**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data)

The Nonlinear Conjugate Gradient (CG) Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.

- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function

Returns
a boolean value indicating successful completion of the optimization algorithm.

bool **cg**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

The Nonlinear Conjugate Gradient (CG) Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns
a boolean value indicating successful completion of the optimization algorithm.

Optimization Control Parameters

The basic control parameters are:

- `fp_t grad_err_tol`: the error tolerance value controlling how small the L_2 norm of the gradient vector $\|\nabla f\|$ should be before 'convergence' is declared.
- `fp_t rel_sol_change_tol`: the error tolerance value controlling how small the proportional change in the solution vector should be before 'convergence' is declared.

  The relative change is computed using:

$$\left\| \frac{x^{(i)} - x^{(i-1)}}{|x^{(i-1)}| + \epsilon} \right\|_1$$

  where ϵ is a small number added for numerical stability.
- `size_t iter_max`: the maximum number of iterations/updates before the algorithm exits.
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.

  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

Additional settings:

- `int cg_settings.method`: Update method.
  - Default value: `2`.
- `fp_t cg_settings.restart_threshold`: parameter ν from step 2 in the algorithm description.
  - Default value: `0.1`.
- `bool use_rel_sol_change_crit`: whether to enable the `rel_sol_change_tol` stopping criterion.
  - Default value: `false`.
- `fp_t cg_settings.wolfe_cons_1`: Line search tuning parameter that controls the tolerance on the Armijo sufficient decrease condition.
  - Default value: `1E-03`.
- `fp_t cg_settings.wolfe_cons_2`: Line search tuning parameter that controls the tolerance on the curvature condition.
  - Default value: `0.10`.
- `int print_level`: Set the level of detail for printing updates on optimization progress.
  - Level `0`: Nothing (default).
  - Level `1`: Print the iteration count and current error values.
  - Level `2`: Level 1 plus the current candidate solution values, $x^{(i+1)}$.
  - Level `3`: Level 2 plus the direction vector, $d^{(i)}$, and the gradient vector, $\nabla_x f(x^{(i+1)})$.
  - Level `4`: Level 3 plus $\beta^{(i)}$.

Examples

Sphere Function

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
sphere_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double obj_val = arma::dot(vals_inp,vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }
```

(continues on next page)

(continued from previous page)

```
    return obj_val;
}

int main()
{
    const int test_dim = 5;

    arma::vec x = arma::ones(test_dim,1); // initial values (1,1,...,1)

    bool success = optim::cg(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "cg: sphere test completed successfully." << "\n";
    } else {
        std::cout << "cg: sphere test completed unsuccessfully." << "\n";
    }

    arma::cout << "cg: solution to sphere test:\n" << x << arma::endl;

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
double
sphere_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double obj_val = vals_inp.dot(vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
    const int test_dim = 5;

    Eigen::VectorXd x = Eigen::VectorXd::Ones(test_dim); // initial values (1,1,...,1)

    bool success = optim::cg(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "cg: sphere test completed successfully." << "\n";
    } else {
        std::cout << "cg: sphere test completed unsuccessfully." << "\n";
```

(continues on next page)

(continued from previous page)

```
    }

    std::cout << "cg: solution to sphere test:\n" << x << std::endl;

    return 0;
}
```

Booth's Function

Code to run this example is given below.

Armadillo Code (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
booth_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    double obj_val = std::pow(x_1 + 2*x_2 - 7.0,2) + std::pow(2*x_1 + x_2 - 5.0,2);

    if (grad_out) {
        (*grad_out)(0) = 10*x_1 + 8*x_2   2*(- 7.0) + 4*(x_2 - 5.0);
        (*grad_out)(1) = 2*(x_1 + 2*x_2 - 7.0)*2 + 2*(2*x_1 + x_2 - 5.0);
    }

    return obj_val;
}

int main()
{
    arma::vec x_2 = arma::zeros(2,1); // initial values (0,0)

    bool success_2 = optim::cg(x, booth_fn, nullptr);

    if (success_2) {
        std::cout << "cg: Booth test completed successfully." << "\n";
    } else {
        std::cout << "cg: Booth test completed unsuccessfully." << "\n";
    }

    arma::cout << "cg: solution to Booth test:\n" << x_2 << arma::endl;

    return 0;
}
```

Eigen Code (Click to show/hide)

```cpp
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
double
booth_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    double obj_val = std::pow(x_1 + 2*x_2 - 7.0,2) + std::pow(2*x_1 + x_2 - 5.0,2);

    if (grad_out) {
        (*grad_out)(0) = 2*(x_1 + 2*x_2 - 7.0) + 2*(2*x_1 + x_2 - 5.0)*2;
        (*grad_out)(1) = 2*(x_1 + 2*x_2 - 7.0)*2 + 2*(2*x_1 + x_2 - 5.0);
    }

    return obj_val;
}

int main()
{
    Eigen::VectorXd x = Eigen::VectorXd::Zero(test_dim); // initial values (0,0)

    bool success_2 = optim::cg(x, booth_fn, nullptr);

    if (success_2) {
        std::cout << "cg: Booth test completed successfully." << "\n";
    } else {
        std::cout << "cg: Booth test completed unsuccessfully." << "\n";
    }

    std::cout << "cg: solution to Booth test:\n" << x_2 << std::endl;

    return 0;
}
```

cg	Conjugate Gradient Method
cg	Conjugate Gradient Method

3.5.4 Gradient Descent

Table of contents

Algorithm Description

The Gradient Descent (GD) algorithm is used solve to optimization problems of the form

$$\min_{x \in X} f(x)$$

where f is convex and (at least) once differentiable.

The updating rule for GD is described below. Let $x^{(i)}$ denote the candidate solution vector at stage i of the algorithm.

1. Compute the descent direction $d^{(i)}$ using one of the methods described below.
3. Update the candidate solution vector using:

$$x^{(i+1)} = x^{(i)} - d^{(i)}$$

The algorithm stops when at least one of the following conditions are met:

1. the norm of the gradient vector, $\|\nabla f\|$, is less than `grad_err_tol`;
2. the relative change between $x^{(i+1)}$ and $x^{(i)}$ is less than `rel_sol_change_tol`;
3. the total number of iterations exceeds `iter_max`.

Gradient Descent Rule

- `gd_settings.method = 0` Vanilla GD:

  $$d^{(i)} = \alpha \times [\nabla_x f(x^{(i)})]$$

  where α, the step size (also known the learning rate), is set by `par_step_size`.
- `gd_settings.method = 1` GD with **momentum**:

  $$d^{(i)} = \mu \times d^{(i-1)} + \alpha \times [\nabla_x f(x^{(i)})]$$

  where μ, the momentum parameter, is set by `par_momentum`.
- `gd_settings.method = 2` Nesterov accelerated gradient descent (**NAG**)

  $$d^{(i)} = \mu \times d^{(i-1)} + \alpha \times \nabla f(x^{(i)} - \mu \times d^{(i-1)})$$

- `gd_settings.method = 3` **AdaGrad**:

  $$d^{(i)} = [\nabla_x f(x^{(i)})] \odot \frac{1}{\sqrt{v^{(i)}} + \epsilon}$$
  $$v^{(i)} = v^{(i-1)} + [\nabla_x f(x^{(i)})] \odot [\nabla_x f(x^{(i)})]$$

- `gd_settings.method = 4` **RMSProp**:

$$d^{(i)} = [\nabla_x f(x^{(i)})] \odot \frac{1}{\sqrt{v^{(i)}} + \epsilon}$$
$$v^{(i)} = \rho \times v^{(i-1)} + (1-\rho) \times [\nabla_x f(x^{(i)})] \odot [\nabla_x f(x^{(i)})]$$

- `gd_settings.method = 5` **AdaDelta**:

$$d^{(i)} = [\nabla_x f(x^{(i)})] \odot \frac{\sqrt{m^{(i)}} + \epsilon}{\sqrt{v^{(i)}} + \epsilon}$$
$$m^{(i)} = \rho \times m^{(i-1)} + (1-\rho) \times [d^{(i-1)}] \odot [d^{(i-1)}]$$
$$v^{(i)} = \rho \times v^{(i-1)} + (1-\rho) \times [\nabla_x f(x^{(i)})] \odot [\nabla_x f(x^{(i)})]$$

- `gd_settings.method = 6` **Adam** (adaptive moment estimation) and **AdaMax**.

$$m^{(i)} = \beta_1 \times m^{(i-1)} + (1-\beta_1) \times [\nabla_x f(x^{(i-1)})]$$
$$v^{(i)} = \beta_2 \times v^{(i-1)} + (1-\beta_2) \times [\nabla_x f(x^{(i)})] \odot [\nabla_x f(x^{(i)})]$$
$$\hat{m} = \frac{m^{(i)}}{1-\beta_1^i}, \;\; \hat{v} = \frac{v^{(i)}}{1-\beta_2^i}$$

  where $m^{(0)} = \mathbf{0}$, and β_1 and β_2 are set by `par_adam_beta_1` and `par_adam_beta_2`, respectively.

  - If `ada_max = false`, then the descent direction is computed as

$$d^{(i)} = \alpha \times \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

  - If `ada_max = true`, then the updating rule for $v^{(i)}$ is no longer based on the L_2 norm; instead

$$v^{(i)} = \max\left\{\beta_2 \times v^{(i-1)}, |\nabla_x f(x^{(i)})|\right\}$$

    The descent direction is computed using

$$d^{(i)} = \alpha \times \frac{\hat{m}}{v^{(i)} + \epsilon}$$

- `gd_settings.method = 7` **Nadam** (adaptive moment estimation) and **NadaMax**

$$m^{(i)} = \beta_1 \times m^{(i-1)} + (1-\beta_1) \times [\nabla_x f(x^{(i-1)})]$$
$$v^{(i)} = \beta_2 \times v^{(i-1)} + (1-\beta_2) \times [\nabla_x f(x^{(i)})] \odot [\nabla_x f(x^{(i)})]$$
$$\hat{m} = \frac{m^{(i)}}{1-\beta_1^i}, \;\; \hat{v} = \frac{v^{(i)}}{1-\beta_2^i}, \;\; \hat{g} = \frac{\nabla_x f(x^{(i)})}{1-\beta_1^i}$$

  where $m^{(0)} = \mathbf{0}$, and β_1 and β_2 are set by `par_adam_beta_1` and `par_adam_beta_2`, respectively.

  - If `ada_max = false`, then the descent direction is computed as

$$d^{(i)} = \alpha \times [\nabla_x f(x^{(i)})] \odot \frac{\beta_1 \hat{m} + (1-\beta_1)\hat{g}}{\sqrt{\hat{v}} + \epsilon}$$

  - If `ada_max = true`, then the updating rule for $v^{(i)}$ is no longer based on the L_2 norm; instead

$$v^{(i)} = \max\left\{\beta_2 \times v^{(i-1)}, |\nabla_x f(x^{(i)})|\right\}$$

    The descent direction is computed using

$$d^{(i)} = \alpha \times [\nabla_x f(x^{(i)})] \odot \frac{\beta_1 \hat{m} + (1-\beta_1)\hat{g}}{v^{(i)} + \epsilon}$$

Function Declarations

bool **gd**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data)

The Gradient Descent Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **gd**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

The Gradient Descent Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns

a boolean value indicating successful completion of the optimization algorithm.

Optimization Control Parameters

The basic control parameters are:

- `fp_t grad_err_tol`: the error tolerance value controlling how small the L_2 norm of the gradient vector $||\nabla f||$ should be before 'convergence' is declared.
- `fp_t rel_sol_change_tol`: the error tolerance value controlling how small the proportional change in the solution vector should be before 'convergence' is declared.

  The relative change is computed using:

$$\left\| \frac{x^{(i)} - x^{(i-1)}}{|x^{(i-1)}| + \epsilon} \right\|_1$$

where ϵ is a small number added for numerical stability.

- `size_t iter_max`: the maximum number of iterations/updates before the algorithm exits.
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

In addition to these:

- `int print_level`: Set the level of detail for printing updates on optimization progress.
  - Level `0`: Nothing (default).
  - Level `1`: Print the current iteration count and error values.
  - Level `2`: Level 1 plus the current candidate solution values, $x^{(i+1)}$.
  - Level `3`: Level 2 plus the direction vector, $d^{(i)}$, and the gradient vector, $\nabla_x f(x^{(i+1)})$.
  - Level `4`: Level 3 plus information about the chosen gradient descent rule.

Examples

Sphere Function

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
sphere_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double obj_val = arma::dot(vals_inp,vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
    const int test_dim = 5;
```

(continues on next page)

(continued from previous page)

```
    arma::vec x = arma::ones(test_dim,1); // initial values (1,1,...,1)

    bool success = optim::gd(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "gd: sphere test completed successfully." << "\n";
    } else {
        std::cout << "gd: sphere test completed unsuccessfully." << "\n";
    }

    arma::cout << "gd: solution to sphere test:\n" << x << arma::endl;

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
double
sphere_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double obj_val = vals_inp.dot(vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
    const int test_dim = 5;

    Eigen::VectorXd x = Eigen::VectorXd::Ones(test_dim); // initial values (1,1,...,1)

    bool success = optim::gd(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "gd: sphere test completed successfully." << "\n";
    } else {
        std::cout << "gd: sphere test completed unsuccessfully." << "\n";
    }

    std::cout << "gd: solution to sphere test:\n" << x << std::endl;

    return 0;
}
```

Booth's Function

Code to run this example is given below.

Armadillo Code (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
booth_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    double obj_val = std::pow(x_1 + 2*x_2 - 7.0,2) + std::pow(2*x_1 + x_2 - 5.0,2);

    if (grad_out) {
        (*grad_out)(0) = 10*x_1 + 8*x_2   2*(- 7.0) + 4*(x_2 - 5.0);
        (*grad_out)(1) = 2*(x_1 + 2*x_2 - 7.0)*2 + 2*(2*x_1 + x_2 - 5.0);
    }

    return obj_val;
}

int main()
{
    arma::vec x_2 = arma::zeros(2,1); // initial values (0,0)

    bool success_2 = optim::gd(x, booth_fn, nullptr);

    if (success_2) {
        std::cout << "gd: Booth test completed successfully." << "\n";
    } else {
        std::cout << "gd: Booth test completed unsuccessfully." << "\n";
    }

    arma::cout << "gd: solution to Booth test:\n" << x_2 << arma::endl;

    return 0;
}
```

Eigen Code (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
double
booth_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);
```

(continues on next page)

(continued from previous page)

```
    double obj_val = std::pow(x_1 + 2*x_2 - 7.0,2) + std::pow(2*x_1 + x_2 - 5.0,2);

    if (grad_out) {
        (*grad_out)(0) = 2*(x_1 + 2*x_2 - 7.0) + 2*(2*x_1 + x_2 - 5.0)*2;
        (*grad_out)(1) = 2*(x_1 + 2*x_2 - 7.0)*2 + 2*(2*x_1 + x_2 - 5.0);
    }

    return obj_val;
}

int main()
{
    Eigen::VectorXd x = Eigen::VectorXd::Zero(test_dim); // initial values (0,0)

    bool success_2 = optim::gd(x, booth_fn, nullptr);

    if (success_2) {
        std::cout << "gd: Booth test completed successfully." << "\n";
    } else {
        std::cout << "gd: Booth test completed unsuccessfully." << "\n";
    }

    std::cout << "gd: solution to Booth test:\n" << x_2 << std::endl;

    return 0;
}
```

gd	Gradient Descent
gd	Gradient Descent

3.5.5 Newton's Method

Table of contents

Algorithm Description

Newton's method is used solve to convex optimization problems of the form

$$\min_{x \in X} f(x)$$

where f is convex and twice differentiable. The algorithm requires both the gradient and Hessian to be known.

The updating rule for Newton's method is described below. Let $x^{(i)}$ denote the candidate solution vector at stage i of the algorithm.

1. Compute the descent direction using:

$$d^{(i)} = -[H(x^{(i)})]^{-1}[\nabla_x f(x^{(i)})]$$

2. Compute the optimal step size using line search:

$$\alpha^{(i)} = \arg\min_{\alpha} f(x^{(i)} + \alpha d^{(i)})$$

3. Update the candidate solution vector using:

$$x^{(i+1)} = x^{(i)} + \alpha^{(i)} d^{(i)}$$

The algorithm stops when at least one of the following conditions are met:

1. the norm of the gradient vector, $\|\nabla f\|$, is less than `grad_err_tol`;
2. the relative change between $x^{(i+1)}$ and $x^{(i)}$ is less than `rel_sol_change_tol`;
3. the total number of iterations exceeds `iter_max`.

Function Declarations

bool **newton**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, Mat_t *hess_out, void *opt_data)> opt_objfn, void *opt_data)

Newton's Nonlinear Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient;
  - `hess_out` a matrix to store the Hessian; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **newton**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, Mat_t *hess_out, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

Newton's Nonlinear Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient;
  - `hess_out` a matrix to store the Hessian; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns

a boolean value indicating successful completion of the optimization algorithm.

Optimization Control Parameters

The basic control parameters are:

- `fp_t grad_err_tol`: the error tolerance value controlling how small the L2 norm of the gradient vector $\|\nabla f\|$ should be before 'convergence' is declared.
- `fp_t rel_sol_change_tol`: the error tolerance value controlling how small the proportional change in the solution vector should be before 'convergence' is declared.

  The relative change is computed using:

$$\left\| \frac{x^{(i)} - x^{(i-1)}}{|x^{(i-1)}| + \epsilon} \right\|_1$$

- `size_t iter_max`: the maximum number of iterations/updates before the algorithm exits.

In addition to these:

- `int print_level`: Set the level of detail for printing updates on optimization progress.
  - Level `0`: Nothing (default).
  - Level `1`: Print the iteration count and current error values.
  - Level `2`: Level 1 plus the current candidate solution values, $x^{(i+1)}$.
  - Level `3`: Level 2 plus the direction vector, $d^{(i)}$, and the gradient vector, $\nabla_x f(x^{(i+1)})$.
  - Level `4`: Level 3 plus the Hessian matrix, $H(x^{(i)})$.

Examples

Example 1

Code to run this example is given below.

Armadillo (Click to show/hide)

```cpp
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
unconstr_test_fn_1_whess(const arma::vec& vals_inp, arma::vec* grad_out, arma::mat* hess_
↪out, void* opt_data)
{
    const double x_1 = vals_inp(0);
    const double x_2 = vals_inp(1);

    double obj_val = 3*x_1*x_1 + 2*x_1*x_2 + x_2*x_2 - 4*x_1 + 5*x_2;

    if (grad_out) {
        (*grad_out)(0) = 6*x_1 + 2*x_2 - 4;
        (*grad_out)(1) = 2*x_1 + 2*x_2 + 5;
    }

    if (hess_out) {
        (*hess_out)(0,0) = 6.0;
        (*hess_out)(0,1) = 2.0;
        (*hess_out)(1,0) = 2.0;
        (*hess_out)(1,1) = 2.0;
    }

    //

    return obj_val;
}

int main()
{
    arma::vec x = arma::zeros(2,1);

    bool success = optim::newton(x, unconstr_test_fn_1_whess, nullptr);

    if (success) {
        std::cout << "newton: test completed successfully." << "\n";
    } else {
        std::cout << "newton: test completed unsuccessfully." << "\n";
    }

    arma::cout << "newton: solution to test:\n" << x << arma::endl;

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
double
unconstr_test_fn_1_whess(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out,␣
↪Eigen::MatrixXd* hess_out, void* opt_data)
{
    const double x_1 = vals_inp(0);
    const double x_2 = vals_inp(1);

    double obj_val = 3*x_1*x_1 + 2*x_1*x_2 + x_2*x_2 - 4*x_1 + 5*x_2;

    if (grad_out) {
        (*grad_out)(0) = 6*x_1 + 2*x_2 - 4;
        (*grad_out)(1) = 2*x_1 + 2*x_2 + 5;
    }

    if (hess_out) {
        (*hess_out)(0,0) = 6.0;
        (*hess_out)(0,1) = 2.0;
        (*hess_out)(1,0) = 2.0;
        (*hess_out)(1,1) = 2.0;
    }

    //

    return obj_val;
}

int main()
{
    Eigen::VectorXd x = Eigen::VectorXd::Zero(2); // initial values (1,1,...,1)

    bool success = optim::newton(x, unconstr_test_fn_1_whess, nullptr);

    if (success) {
        std::cout << "newton: test completed successfully." << "\n";
    } else {
        std::cout << "newton: test completed unsuccessfully." << "\n";
    }

    std::cout << "newton: solution to test:\n" << x << std::endl;

    return 0;
}
```

newton	Newton
newton	Newton

3.6 Simplex-based Optimization

3.6.1 Nelder-Mead

Table of contents

Algorithm Description

Nelder-Mead is a derivative-free simplex method, used solve to optimization problems of the form

$$\min_{x \in X} f(x)$$

where f need not be convex or differentiable.

The updating rule for Nelder-Mead is described below. Let $x^{(i)}$ denote the simplex values at stage i of the algorithm.

1. Sort the simplex vertices x in order of function values, from smallest to largest:

$$f(x^{(i)})(1,:) \leq f(x^{(i)})(2,:) \leq \cdots \leq f(x^{(i)})(n+1,:)$$

2. Calculate the centroid value up to the n th vertex:

$$\bar{x} = \frac{1}{n} \sum_{j=1}^{n} x^{(i)}(j,:)$$

   and compute the reflection point:

$$x^r = \bar{x} + \alpha(\bar{x} - x^{(i)}(n+1,:))$$

   where α is set by `par_alpha`.

   If $f(x^{(i)}(1,:)) \leq f(x^r) < f(x^{(i)}(n,:))$, then

$$x^{(i+1)}(n+1,:) = x^r, \quad \textbf{and go to Step 1.}$$

   Otherwise continue to Step 3.

3. If $f(x^r) \geq f(x^{(i)}(1,:))$ then go to Step 4, otherwise compute the expansion point:

$$x^e = \bar{x} + \gamma(x^r - \bar{x})$$

   where γ is set by `par_gamma`.

Set

$$x^{(i+1)}(n+1,:) = \begin{cases} x^e & \text{if } f(x^e) < f(x^r) \\ x^r & \text{else} \end{cases}$$

and go to Step 1.

4. If $f(x^r) < f(x^{(i)}(n,:))$ then compute the outside or inside contraction:

$$x^c = \begin{cases} \bar{x} + \beta(x^r - \bar{x}) & \text{if } f(x^r) < f(x^{(i)}(n+1,:)) \\ \bar{x} - \beta(x^r - \bar{x}) & \text{else} \end{cases}$$

If $f(x^c) < f(x^{(i)}(n+1,:))$, then

$$x^{(i+1)}(n+1,:) = x^c, \quad \textbf{and go to Step 1.}$$

Otherwise go to Step 5.

5. Shrink the simplex toward $x^{(i)}(1,:)$:

$$x^{(i+1)}(j,:) = x^{(i)}(1,:) + \delta(x^{(i)}(j,:) - x^{(i)}(1,:)), \;\; j = 2, \ldots, n+1$$

where δ is set by `par_delta`. Go to Step 1.

The algorithm stops when at least one of the following conditions are met:

1. the relative change in the simplex of function values, defined as:

$$\frac{\max\{|f(x^{(i+1)}(1,:)) - f(x^{(i)}(1,:))|, |f(x^{(i+1)}(n+1,:)) - f(x^{(i)}(1,:))|\}}{\max_j |f(x^{(i+1)}(j,:))| + \epsilon};$$

is less than `rel_objfn_change_tol`.

2. the relative change between $x^{(i+1)}$ and $x^{(i)}$ is less than `rel_sol_change_tol`;
3. the total number of iterations exceeds `iter_max`.

Function Declarations

bool **nm**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data)

The Nelder-Mead Simplex-based Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **nm**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

The Nelder-Mead Simplex-based Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns

a boolean value indicating successful completion of the optimization algorithm.

Optimization Control Parameters

The basic control parameters are:

- `fp_t rel_objfn_change_tol`: the error tolerance value controlling how small the relative change in the simplex of function values, defined as:

$$\frac{\max\{|f(x^{(i+1)}(1,:)) - f(x^{(i)}(1,:))|, |f(x^{(i+1)}(n+1,:)) - f(x^{(i)}(1,:))|\}}{\max_j |f(x^{(i+1)}(j,:))| + \epsilon};$$

  should be before 'convergence' is declared.

- `fp_t rel_sol_change_tol`: the error tolerance value controlling how small the proportional change in the solution vector should be before 'convergence' is declared.

  The relative change is computed using:

$$\frac{\max_{j,k} |x^{(i+1)}(j,k) - x^{(i)}(j,k)|}{\max_{j,k} |x^{(i)}(j,k)| + \epsilon}$$

  where ϵ is a small number added for numerical stability.

- `size_t iter_max`: the maximum number of iterations/updates before the algorithm exits.
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.
- `struct nm_settings_t`, which defines several parameters that control the behavior of the simplex.

  - `bool adaptive_pars = true`: scale the contraction, expansion, and shrinkage parameters using the dimension of the optimization problem.
  - `fp_t par_alpha = 1.0`: reflection parameter.
  - `fp_t par_beta = 0.5`: contraction parameter.
  - `fp_t par_gamma = 2.0`: expansion parameter.
  - `fp_t par_delta = 0.5`: shrinkage parameter.
  - `bool custom_initial_simplex = false`: whether to use user-defined values for the initial simplex matrix.
  - `Mat_t initial_simplex_points`: user-defined values for the initial simplex (optional). Dimensions: $(n+1) \times n$.

In addition to these:

- `int print_level`: Set the level of detail for printing updates on optimization progress.
  - Level 1: Print the iteration count and current error values.
  - Level 2: Level 1 plus the current candidate solution values.
  - Level 3: Level 2 plus the simplex matrix, $x^{(i)}$, and value of the objective function at each vertex of the simplex.

Examples

Sphere Function

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
sphere_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double obj_val = arma::dot(vals_inp,vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
    const int test_dim = 5;
```

(continues on next page)

(continued from previous page)

```
    arma::vec x = arma::ones(test_dim,1); // initial values (1,1,...,1)

    bool success = optim::nm(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "nm: sphere test completed successfully." << "\n";
    } else {
        std::cout << "nm: sphere test completed unsuccessfully." << "\n";
    }

    arma::cout << "nm: solution to sphere test:\n" << x << arma::endl;

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
double
sphere_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double obj_val = vals_inp.dot(vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
    const int test_dim = 5;

    Eigen::VectorXd x = Eigen::VectorXd::Ones(test_dim); // initial values (1,1,...,1)

    bool success = optim::nm(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "nm: sphere test completed successfully." << "\n";
    } else {
        std::cout << "nm: sphere test completed unsuccessfully." << "\n";
    }

    std::cout << "nm: solution to sphere test:\n" << x << std::endl;

    return 0;
}
```

Booth's Function

Code to run this example is given below.

Armadillo Code (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
double
booth_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    double obj_val = std::pow(x_1 + 2*x_2 - 7.0,2) + std::pow(2*x_1 + x_2 - 5.0,2);

    if (grad_out) {
        (*grad_out)(0) = 10*x_1 + 8*x_2   2*(- 7.0) + 4*(x_2 - 5.0);
        (*grad_out)(1) = 2*(x_1 + 2*x_2 - 7.0)*2 + 2*(2*x_1 + x_2 - 5.0);
    }

    return obj_val;
}

int main()
{
    arma::vec x_2 = arma::zeros(2,1); // initial values (0,0)

    bool success_2 = optim::nm(x, booth_fn, nullptr);

    if (success_2) {
        std::cout << "nm: Booth test completed successfully." << "\n";
    } else {
        std::cout << "nm: Booth test completed unsuccessfully." << "\n";
    }

    arma::cout << "nm: solution to Booth test:\n" << x_2 << arma::endl;

    return 0;
}
```

Eigen Code (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
double
booth_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);
```

(continues on next page)

(continued from previous page)

```
    double obj_val = std::pow(x_1 + 2*x_2 - 7.0,2) + std::pow(2*x_1 + x_2 - 5.0,2);

    if (grad_out) {
        (*grad_out)(0) = 2*(x_1 + 2*x_2 - 7.0) + 2*(2*x_1 + x_2 - 5.0)*2;
        (*grad_out)(1) = 2*(x_1 + 2*x_2 - 7.0)*2 + 2*(2*x_1 + x_2 - 5.0);
    }

    return obj_val;
}

int main()
{
    Eigen::VectorXd x = Eigen::VectorXd::Zero(test_dim); // initial values (0,0)

    bool success_2 = optim::nm(x, booth_fn, nullptr);

    if (success_2) {
        std::cout << "nm: Booth test completed successfully." << "\n";
    } else {
        std::cout << "nm: Booth test completed unsuccessfully." << "\n";
    }

    std::cout << "nm: solution to Booth test:\n" << x_2 << std::endl;

    return 0;
}
```

nm	Nelder-Mead
nm	Nelder-Mead

3.7 Metaheuristic Optimization

3.7.1 Differential Evolution

Table of contents

Algorithm Description

Differential Evolution (DE) is a stochastic genetic search algorithm for global optimization of problems of the form

$$\min_{x \in X} f(x)$$

where f is potentially ill-behaved in one or more ways, such as non-convexity and/or non-differentiability. The updating rule for DE is described below.

Let $X^{(i)}$ denote the $N \times d$ dimensional array of input values at stage i of the algorithm, where each row corresponds to a different vector of candidate solutions.

1. The Mutation Step. For unique random indices $a, b, c \in \{1, \dots, d\}$, set the mutation proposal $X^{(*)}$ as follows.
    1. If `de_mutation_method = 1`, use the 'rand' method:

    $$X^{(*)} = X^{(i)}(c, :) + F \times \Big(X^{(i)}(a, :) - X^{(i)}(b, :) \Big)$$

    where F is the mutation parameter, set via `de_par_F`.
    2. If `de_mutation_method = 2`, use the 'best' method:

    $$X^{(*)} = X^{(i)}(\text{best}, :) + F \times (X^{(i)}(a, :) - X^{(i)}(b, :))$$

    where

    $$X^{(i)}(\text{best}, :) := \arg\min \Big\{ f(X^{(i)}(1, :)), \dots, f(X^{(i)}(N, :)) \Big\}$$

2. The Crossover Step.
    1. Choose a random integer $r_k \in \{1, \dots, d\}$.
    2. Draw a vector u of independent uniform random variables of length d
    3. For each $j \in \{1, \dots, N\}$ and $k \in \{1, \dots, d\}$, set

    $$X_c^{(*)}(j, k) = \begin{cases} X^*(j, k) & \text{if } u_k \leq CR \text{ or } k = r_k \\ X^{(i)}(j, k) & \text{else} \end{cases}$$

    where $CR \in [0, 1]$ is the crossover parameter, set via `de_par_CR`.
3. The Update Step.

$$X^{(i+1)}(j, :) = \begin{cases} X_c^*(j, :) & \text{if } f(X_c^*(j, :)) < f(X^{(i)}(j, :)) \\ X^{(i)}(j, :) & \text{else} \end{cases}$$

The algorithm stops when at least one of the following conditions are met:

1. the relative improvement in the objective function from the best candidate solution is less than `rel_objfn_change_tol` between `de_settings.check_freq` number of generations;
2. the total number of generations exceeds `de_settings.n_gen`.

DE with Population Reduction and Multiple Mutation Strategies

TBW.

Function Declarations

bool **de**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data)

The Differential Evolution (DE) Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **de**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

The Differential Evolution (DE) Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns

a boolean value indicating successful completion of the optimization algorithm.

---

bool **de_prmm**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data)

The Differential Evolution (DE) with Population Reduction and Multiple Mutation Strategies (PRMM) Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.

Returns
a boolean value indicating successful completion of the optimization algorithm.

bool **de_prmm**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

The Differential Evolution (DE) with Population Reduction and Multiple Mutation Strategies (PRMM) Optimization Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns
a boolean value indicating successful completion of the optimization algorithm.

Optimization Control Parameters

The basic control parameters are:

- `fp_t rel_objfn_change_tol`: the error tolerance value controlling how small the relative change in best candidate solution should be before 'convergence' is declared.
- `size_t iter_max`: the maximum number of iterations/updates before the algorithm exits.
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

In addition to these:

- `int print_level`: Set print level.
  - Level 1: Print iteration count and error value.
  - Level 2: Level 1 and print best input values and corresponding objective function value.

- Level 3: Level 2 and print full population matrix, X.

Examples

Ackley Function

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

#define OPTIM_PI 3.14159265358979

double
ackley_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    const double x = vals_inp(0);
    const double y = vals_inp(1);

    const double obj_val = 20 + std::exp(1) - 20*std::exp( -0.2*std::sqrt(0.5*(x*x +␣
↪y*y)) ) - std::exp( 0.5*(std::cos(2 * OPTIM_PI * x) + std::cos(2 * OPTIM_PI * y)) );

    return obj_val;
}

int main()
{
    arma::vec x = arma::ones(2,1) + 1.0; // initial values: (2,2)

    bool success = optim::de(x, ackley_fn, nullptr);

    if (success) {
        std::cout << "de: Ackley test completed successfully." << std::endl;
    } else {
        std::cout << "de: Ackley test completed unsuccessfully." << std::endl;
    }

    arma::cout << "de: solution to Ackley test:\n" << x << arma::endl;

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

#define OPTIM_PI 3.14159265358979
```

(continues on next page)

(continued from previous page)

```
double
ackley_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    const double x = vals_inp(0);
    const double y = vals_inp(1);

    const double obj_val = 20 + std::exp(1) - 20*std::exp( -0.2*std::sqrt(0.5*(x*x +␣
↪y*y)) ) - std::exp( 0.5*(std::cos(2 * OPTIM_PI * x) + std::cos(2 * OPTIM_PI * y)) );

    return obj_val;
}

int main()
{
    Eigen::VectorXd x = 2.0 * Eigen::VectorXd::Ones(2); // initial values: (2,2)

    bool success = optim::de(x, ackley_fn, nullptr);

    if (success) {
        std::cout << "de: Ackley test completed successfully." << std::endl;
    } else {
        std::cout << "de: Ackley test completed unsuccessfully." << std::endl;
    }

    arma::cout << "de: solution to Ackley test:\n" << x << arma::endl;

    return 0;
}
```

Rastrigin Function

Code to run this example is given below.

Armadillo Code (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

#define OPTIM_PI 3.14159265358979

struct rastrigin_fn_data {
    double A;
};

double
rastrigin_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    const int n = vals_inp.n_elem;

    rastrigin_fn_data* objfn_data = reinterpret_cast<rastrigin_fn_data*>(opt_data);
```

(continues on next page)

(continued from previous page)

```
    const double A = objfn_data->A;

    double obj_val = A*n + arma::accu( arma::pow(vals_inp,2) - A*arma::cos(2 * OPTIM_PI␣
↪* vals_inp) );

    return obj_val;
}

int main()
{
    rastrigin_fn_data test_data;
    test_data.A = 10;

    arma::vec x = arma::ones(2,1) + 1.0; // initial values: (2,2)

    bool success = optim::de(x, rastrigin_fn, &test_data);

    if (success) {
        std::cout << "de: Rastrigin test completed successfully." << std::endl;
    } else {
        std::cout << "de: Rastrigin test completed unsuccessfully." << std::endl;
    }

    arma::cout << "de: solution to Rastrigin test:\n" << x << arma::endl;

    return 0;
}
```

Eigen Code (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

#define OPTIM_PI 3.14159265358979

struct rastrigin_fn_data {
    double A;
};

double
rastrigin_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    const int n = vals_inp.n_elem;

    rastrigin_fn_data* objfn_data = reinterpret_cast<rastrigin_fn_data*>(opt_data);
    const double A = objfn_data->A;

    double obj_val = A*n + vals_inp.array().pow(2).sum() - A * (2 * OPTIM_PI * vals_inp).
↪array().cos().sum();

    return obj_val;
}
```

(continues on next page)

(continued from previous page)

```
int main()
{
    rastrigin_fn_data test_data;
    test_data.A = 10;

    Eigen::VectorXd x = 2.0 * Eigen::VectorXd::Ones(2); // initial values: (2,2)

    bool success = optim::de(x, rastrigin_fn, &test_data);

    if (success) {
        std::cout << "de: Rastrigin test completed successfully." << std::endl;
    } else {
        std::cout << "de: Rastrigin test completed unsuccessfully." << std::endl;
    }

    arma::cout << "de: solution to Rastrigin test:\n" << x << arma::endl;

    return 0;
}
```

de	Differential Evolution
de	Differential Evolution

de-prmm	DE with Population Reduction and Multiple Mutation (PRMM) Strategies
de-prmm	DE with Population Reduction and Multiple Mutation (PRMM) Strategies

3.7.2 Particle Swarm Optimization

Table of contents

Algorithm Description

Particle Swarm Optimization (PSO) is a stochastic swarm intelligence algorithm for global optimization

$$\min_{x \in X} f(x)$$

where f is potentially ill-behaved in one or more ways, such as non-convexity and/or non-differentiability. The updating rule for PSO is described below.

Let $X^{(i)}$ denote the $N \times d$ dimensional array of input values at stage i of the algorithm, where each row corresponds to a different vector of candidate solutions.

1. Update the velocity and position matrices. Sample two d-dimensional iid uniform random vectors, R_C, R_S.

   Update each velocity vector using:

   $$V^{(i+1)}(j.:) = wV^{(i+1)}(j,:) + c_C \times R_C \odot (X_b^{(i)}(j,:) - X^{(i)}(j,:)) + c_S \times R_S \odot (g_b - X^{(i)}(j,:))$$

   Each position vector is updated using:

   $$X^{(i+1)}(j,:) = X^{(i)}(j,:) + V^{(i+1)}(j,:)$$

2. Update local-best particle.

   $$X_b^{(i+1)}(j,:) = \begin{cases} X^{(i+1)}(j,:) & \text{if } f(X^{(i+1)}(j,:)) < f(X_b^{(i)}(j,:)) \\ X_b^{(i)}(j,:) & \text{else} \end{cases}$$

3. Update the global-best particle.

   Let

   $$j^{(*)} = \arg \min_{j \in \{1,\dots,N\}} f(X^{(i+1)}(j,:))$$

   Then

   $$g_b = \begin{cases} X^{(i+1)}(j^{(*)},:) & \text{if } f(X^{(i+1)}(j^{(*)},:)) < f(g_b) \\ g_b & \text{else} \end{cases}$$

The algorithm stops when at least one of the following conditions are met:

1. the relative improvement in the objective function is less than `rel_objfn_change_tol` between `pso_settings.check_freq` number of generations;
2. the total number of generations exceeds `pso_settings.n_gen`.

PSO with Differentially-Perturbed Velocity

TBW.

Function Declarations

bool **pso**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data)

Particle Swarm Optimization (PSO) Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **pso**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

Particle Swarm Optimization (PSO) Algorithm.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns

a boolean value indicating successful completion of the optimization algorithm.

---

bool **pso_dv**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data)

Particle Swarm Optimization (PSO) with Differentially-Perturbed Velocity (DV)

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.

- **opt_data** – additional data passed to the user-provided function.

Returns
a boolean value indicating successful completion of the optimization algorithm.

bool **pso_dv**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

Particle Swarm Optimization (PSO) with Differentially-Perturbed Velocity (DV)

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns
a boolean value indicating successful completion of the optimization algorithm.

Optimization Control Parameters

The basic control parameters are:

- `fp_t rel_objfn_change_tol`: the error tolerance value controlling how small the relative change in best candidate solution should be before 'convergence' is declared.
- `size_t iter_max`: the maximum number of iterations/updates before the algorithm exits.
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

In addition to these:

- `int print_level`: Set print level.
  - Level 1: Print iteration count and error value.
  - Level 2: Level 1 and print best input values, as well as objective function values.
  - Level 3: Level 2 and print full matrix X.

Examples

Ackley Function

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

#define OPTIM_PI 3.14159265358979

double
ackley_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    const double x = vals_inp(0);
    const double y = vals_inp(1);

    double obj_val = 20 + std::exp(1) - 20*std::exp( -0.2*std::sqrt(0.5*(x*x + y*y)) ) -␣
↪std::exp( 0.5*(std::cos(2 * OPTIM_PI * x) + std::cos(2 * OPTIM_PI * y)) );

    return obj_val;
}

int main()
{
    arma::vec x = arma::ones(2,1) + 1.0; // initial values: (2,2)

    bool success = optim::pso(x, ackley_fn, nullptr);

    if (success) {
        std::cout << "pso: Ackley test completed successfully." << std::endl;
    } else {
        std::cout << "pso: Ackley test completed unsuccessfully." << std::endl;
    }

    arma::cout << "pso: solution to Ackley test:\n" << x << arma::endl;

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

#define OPTIM_PI 3.14159265358979

double
ackley_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    const double x = vals_inp(0);
    const double y = vals_inp(1);
```

(continues on next page)

(continued from previous page)

```
    double obj_val = 20 + std::exp(1) - 20*std::exp( -0.2*std::sqrt(0.5*(x*x + y*y)) ) -␣
↪std::exp( 0.5*(std::cos(2 * OPTIM_PI * x) + std::cos(2 * OPTIM_PI * y)) );

    return obj_val;
}

int main()
{
    Eigen::VectorXd x = 2.0 * Eigen::VectorXd::Ones(2); // initial values: (2,2)

    bool success = optim::pso(x, ackley_fn, nullptr);

    if (success) {
        std::cout << "pso: Ackley test completed successfully." << std::endl;
    } else {
        std::cout << "pso: Ackley test completed unsuccessfully." << std::endl;
    }

    arma::cout << "pso: solution to Ackley test:\n" << x << arma::endl;

    return 0;
}
```

Rastrigin Function

Code to run this example is given below.

Armadillo Code (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

#define OPTIM_PI 3.14159265358979

struct rastrigin_fn_data {
    double A;
};

double
rastrigin_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    const int n = vals_inp.n_elem;

    rastrigin_fn_data* objfn_data = reinterpret_cast<rastrigin_fn_data*>(opt_data);
    const double A = objfn_data->A;

    double obj_val = A*n + arma::accu( arma::pow(vals_inp,2) - A*arma::cos(2 * OPTIM_PI␣
↪* vals_inp) );
```

(continues on next page)

(continued from previous page)

```
    return obj_val;
}

int main()
{
    rastrigin_fn_data test_data;
    test_data.A = 10;

    arma::vec x = arma::ones(2,1) + 1.0; // initial values: (2,2)

    bool success = optim::pso(x, rastrigin_fn, &test_data);

    if (success) {
        std::cout << "pso: Rastrigin test completed successfully." << std::endl;
    } else {
        std::cout << "pso: Rastrigin test completed unsuccessfully." << std::endl;
    }

    arma::cout << "pso: solution to Rastrigin test:\n" << x << arma::endl;

    return 0;
}
```

Eigen Code (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

#define OPTIM_PI 3.14159265358979

struct rastrigin_fn_data {
    double A;
};

double
rastrigin_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    const int n = vals_inp.n_elem;

    rastrigin_fn_data* objfn_data = reinterpret_cast<rastrigin_fn_data*>(opt_data);
    const double A = objfn_data->A;

    double obj_val = A*n + vals_inp.array().pow(2).sum() - A * (2 * OPTIM_PI * vals_inp).
↪array().cos().sum();

    return obj_val;
}

int main()
{
    rastrigin_fn_data test_data;
    test_data.A = 10;
```

(continues on next page)

(continued from previous page)

```
    Eigen::VectorXd x = 2.0 * Eigen::VectorXd::Ones(2); // initial values: (2,2)

    bool success = optim::pso(x, rastrigin_fn, &test_data);

    if (success) {
        std::cout << "pso: Rastrigin test completed successfully." << std::endl;
    } else {
        std::cout << "pso: Rastrigin test completed unsuccessfully." << std::endl;
    }

    arma::cout << "pso: solution to Rastrigin test:\n" << x << arma::endl;

    return 0;
}
```

pso	Particle Swarm Optimization (PSO)
pso	Particle Swarm Optimization (PSO)

pso-dv	PSO with Differentially-Perturbed Velocity (DV)
pso-dv	PSO with Differentially-Perturbed Velocity (DV)

3.8 Constrained Optimization

3.8.1 Sequential Unconstrained Minimization Technique

Table of contents

Description

For a general problem

$$\min_x f(x) \text{ subject to } g_k(x) \leq 0, \;\; k \in \{1, \ldots, K\}$$

The Sequential Unconstrained Minimization Technique solves:

$$\min_x \left\{ f(x) + c(i) \times \frac{1}{2} \sum_{k=1}^{K} \left(\max\{0, g_k(x)\} \right)^2 \right\}$$

The algorithm stops when the error is less than `err_tol`, or the total number of 'generations' exceeds a desired (or default) value.

Definitions

bool **sumt**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, std::function<ColVec_t(const ColVec_t &vals_inp, Mat_t *jacob_out, void *constr_data)> constr_fn, void *constr_data)

Sequential Unconstrained Minimization Technique.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **constr_fn** – the constraint functions, in vector form, taking three arguments.
- **constr_data** – additional data passed to the constraints functions.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **sumt**(ColVec_t &init_out_vals, std::function<fp_t(const ColVec_t &vals_inp, ColVec_t *grad_out, void *opt_data)> opt_objfn, void *opt_data, std::function<ColVec_t(const ColVec_t &vals_inp, Mat_t *jacob_out, void *constr_data)> constr_fn, void *constr_data, algo_settings_t &settings)

Sequential Unconstrained Minimization Technique.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs;
  - `grad_out` a vector to store the gradient; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **constr_fn** – the constraint functions, in vector form, taking three arguments.
- **constr_data** – additional data passed to the constraints functions.
- **settings** – parameters controlling the optimization routine.

Returns

a boolean value indicating successful completion of the optimization algorithm.

Examples

sumt	Sequential Unconstrained Minimization Technique
sumt	Sequential Unconstrained Minimization Technique

3.9 Root Finding

3.9.1 Broyden

Table of contents

Algorithm Description

Broyden's method is an algorithm for solving systems of nonlinear equations of the form:

$$F(x^{(*)}) = \mathbf{0}$$

where $F : \mathbb{R}^n \to \mathbb{R}^m$ is convex and differentiable.

The updating rule for Broyden's method is described below. Let $x^{(i)}$ denote the function input values at stage i of the algorithm.

1. Compute the descent direction using:

$$d^{(i)} = -B^{(i)} F(x^{(i)})$$

   where B is an approximation to the inverse Jacobian matrix (the calculation of which is described in step 3).

2. Update the candidate solution vector using:

$$x^{(i+1)} = x^{(i)} + d^{(i)}$$

3. Update the approximate inverse Jacobian matrix, B, using:

$$B^{(i+1)} = B^{(i)} + \frac{1}{[y^{(i+1)}]^\top y^{(i+1)}} (s^{(i+1)} - B^{(i)} y^{(i+1)})[y^{(i+1)}]^\top$$

where

$$s^{(i)} := x^{(i)} - x^{(i-1)}$$
$$y^{(i)} := F(x^{(i)}) - F(x^{(i-1)})$$

The algorithm stops when at least one of the following conditions are met:

1. $\|F\|$ is less than `rel_objfn_change_tol`.
2. the relative change between $x^{(i+1)}$ and $x^{(i)}$ is less than `rel_sol_change_tol`;
3. the total number of iterations exceeds `iter_max`.

Function Declarations

bool **broyden**(ColVec_t &init_out_vals, std::function<ColVec_t(const ColVec_t &vals_inp, void *opt_data)> opt_objfn, void *opt_data)

Broyden's method for solving systems of nonlinear equations, without Jacobian.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.

Returns
a boolean value indicating successful completion of the optimization algorithm.

bool **broyden**(ColVec_t &init_out_vals, std::function<ColVec_t(const ColVec_t &vals_inp, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

Broyden's method for solving systems of nonlinear equations, without Jacobian.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns
a boolean value indicating successful completion of the optimization algorithm.

bool **broyden**(ColVec_t &init_out_vals, std::function<ColVec_t(const ColVec_t &vals_inp, void *opt_data)> opt_objfn, void *opt_data, std::function<Mat_t(const ColVec_t &vals_inp, void *jacob_data)> jacob_objfn, void *jacob_data)

Broyden's method for solving systems of nonlinear equations, with Jacobian.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **jacob_objfn** – a function to calculate the Jacobian matrix, taking two arguments:
  - `vals_inp` a vector of inputs; and
  - `jacob_data` additional data passed to the Jacobian function.
- **jacob_data** – additional data passed to the Jacobian function.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **broyden**(ColVec_t &init_out_vals, std::function<ColVec_t(const ColVec_t &vals_inp, void *opt_data)> opt_objfn, void *opt_data, std::function<Mat_t(const ColVec_t &vals_inp, void *jacob_data)> jacob_objfn, void *jacob_data, algo_settings_t &settings)

Broyden's method for solving systems of nonlinear equations, with Jacobian.

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **jacob_objfn** – a function to calculate the Jacobian matrix, taking two arguments:
  - `vals_inp` a vector of inputs; and
  - `jacob_data` additional data passed to the Jacobian function.
- **jacob_data** – additional data passed to the Jacobian function.
- **settings** – parameters controlling the optimization routine.

Returns

a boolean value indicating successful completion of the optimization algorithm.

Optimization Control Parameters

The basic control parameters are:

- `fp_t rel_objfn_change_tol`: the error tolerance value controlling how small $\|F\|$ should be before 'convergence' is declared.
- `fp_t rel_sol_change_tol`: the error tolerance value controlling how small the proportional change in the solution vector should be before 'convergence' is declared.

  The relative change is computed using:

$$\left\| \frac{x^{(i)} - x^{(i-1)}}{|x^{(i-1)}| + \epsilon} \right\|_1$$

  where ϵ is a small number added for numerical stability.
- `size_t iter_max`: the maximum number of iterations/updates before the algorithm exits.
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

In addition to these:

- `int print_level`: Set the level of detail for printing updates on optimization progress.
  - Level `0`: Nothing (default).
  - Level `1`: Print the current iteration count and error values.
  - Level `2`: Level 1 plus the current candidate solution values, $x^{(i+1)}$.
  - Level `3`: Level 2 plus the direction vector, $d^{(i)}$, and the function values, $F(x^{(i+1)})$.
  - Level `4`: Level 3 plus the components used to update the approximate inverse Jacobian matrix: $s^{(i+1)}$, $y^{(i+1)}$, and $B^{(i+1)}$.

Examples

Example 1

$$F(\mathbf{x}) = \begin{bmatrix} \exp(-\exp(-(x_1 + x_2))) - x_2(1 + x_1^2) \\ x_1 \cos(x_2) + x_2 \sin(x_1) - 0.5 \end{bmatrix}$$

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
```

(continues on next page)

(continued from previous page)

```
arma::vec
zeros_test_objfn_1(const arma::vec& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    arma::vec ret(2);

    ret(0) = std::exp(-std::exp(-(x_1+x_2))) - x_2*(1 + std::pow(x_1,2));
    ret(1) = x_1*std::cos(x_2) + x_2*std::sin(x_1) - 0.5;

    //

    return ret;
}

inline
arma::mat
zeros_test_jacob_1(const arma::vec& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    arma::mat ret(2,2);

    ret(0,0) = std::exp(-std::exp(-(x_1+x_2))-(x_1+x_2)) - 2*x_1*x_1;
    ret(0,1) = std::exp(-std::exp(-(x_1+x_2))-(x_1+x_2)) - x_1*x_1 - 1.0;
    ret(1,0) = std::cos(x_2) + x_2*std::cos(x_1);
    ret(1,1) = -x_1*std::sin(x_2) + std::cos(x_1);

    //

    return ret;
}

int main()
{
    arma::vec x = arma::zeros(2,1); // initial values (0,0)

    bool success = optim::broyden(x, zeros_test_objfn_1, nullptr);

    if (success) {
        std::cout << "broyden: test_1 completed successfully." << "\n";
    } else {
        std::cout << "broyden: test_1 completed unsuccessfully." << "\n";
    }

    arma::cout << "broyden: solution to test_1:\n" << x << arma::endl;
```

(continues on next page)

(continued from previous page)

```
    //

    x = arma::zeros(2,1);

    success = optim::broyden(x, zeros_test_objfn_1, nullptr, zeros_test_jacob_1,␣
↪nullptr);

    if (success) {
        std::cout << "broyden with jacobian: test_1 completed successfully." << "\n";
    } else {
        std::cout << "broyden with jacobian: test_1 completed unsuccessfully." << "\n";
    }

    arma::cout << "broyden with jacobian: solution to test_1:\n" << x << arma::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
Eigen::VectorXd
zeros_test_objfn_1(const Eigen::VectorXd& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    Eigen::VectorXd ret(2);

    ret(0) = std::exp(-std::exp(-(x_1+x_2))) - x_2*(1 + std::pow(x_1,2));
    ret(1) = x_1*std::cos(x_2) + x_2*std::sin(x_1) - 0.5;

    //

    return ret;
}

inline
Eigen::MatrixXd
zeros_test_jacob_1(const Eigen::VectorXd& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);
```

(continues on next page)

(continued from previous page)

```
    //

    Eigen::MatrixXd ret(2,2);

    ret(0,0) = std::exp(-std::exp(-(x_1+x_2))-(x_1+x_2)) - 2*x_1*x_1;
    ret(0,1) = std::exp(-std::exp(-(x_1+x_2))-(x_1+x_2)) - x_1*x_1 - 1.0;
    ret(1,0) = std::cos(x_2) + x_2*std::cos(x_1);
    ret(1,1) = -x_1*std::sin(x_2) + std::cos(x_1);

    //

    return ret;
}

int main()
{
    Eigen::VectorXd x = Eigen::VectorXd::Zero(2); // initial values (0,0)

    bool success = optim::broyden(x, zeros_test_objfn_1, nullptr);

    if (success) {
        std::cout << "broyden: test_1 completed successfully." << "\n";
    } else {
        std::cout << "broyden: test_1 completed unsuccessfully." << "\n";
    }

    std::cout << "broyden: solution to test_1:\n" << x << std::endl;

    //

    x = Eigen::VectorXd::Zero(2);

    success = optim::broyden(x, zeros_test_objfn_1, nullptr, zeros_test_jacob_1,␣
↪nullptr);

    if (success) {
        std::cout << "broyden with jacobian: test_1 completed successfully." << "\n";
    } else {
        std::cout << "broyden with jacobian: test_1 completed unsuccessfully." << "\n";
    }

    std::cout << "broyden with jacobian: solution to test_1:\n" << x << std::endl;

    //

    return 0;
}
```

Example 2

$$F(\mathbf{x}) = \begin{bmatrix} 2x_1 - x_2 - \exp(-x_1) \\ -x_1 + 2x_2 - \exp(-x_2) \end{bmatrix}$$

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
arma::vec
zeros_test_objfn_2(const arma::vec& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    arma::vec ret(2);

    ret(0) =   2*x_1 - x_2   - std::exp(-x_1);
    ret(1) = - x_1   + 2*x_2 - std::exp(-x_2);

    //

    return ret;
}

inline
arma::mat
zeros_test_jacob_2(const arma::vec& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    arma::mat ret(2,2);

    ret(0,0) = 2 + std::exp(-x_1);
    ret(0,1) = - 1.0;
    ret(1,0) = - 1.0;
    ret(1,1) = 2 + std::exp(-x_2);

    //

    return ret;
}

int main()
```

(continues on next page)

(continued from previous page)

```
{
    arma::vec x = arma::zeros(2,1); // initial values (0,0)

    bool success = optim::broyden(x, zeros_test_objfn_2, nullptr);

    if (success) {
        std::cout << "broyden: test_2 completed successfully." << "\n";
    } else {
        std::cout << "broyden: test_2 completed unsuccessfully." << "\n";
    }

    arma::cout << "broyden: solution to test_2:\n" << x << arma::endl;

    //

    x = arma::zeros(2,1);

    success = optim::broyden(x, zeros_test_objfn_2, nullptr, zeros_test_jacob_2,␣
↪nullptr);

    if (success) {
        std::cout << "broyden with jacobian: test_2 completed successfully." << "\n";
    } else {
        std::cout << "broyden with jacobian: test_2 completed unsuccessfully." << "\n";
    }

    arma::cout << "broyden with jacobian: solution to test_2:\n" << x << arma::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
Eigen::VectorXd
zeros_test_objfn_2(const Eigen::VectorXd& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    Eigen::VectorXd ret(2);

    ret(0) =   2*x_1 - x_2   - std::exp(-x_1);
    ret(1) = - x_1   + 2*x_2 - std::exp(-x_2);
```

(continues on next page)

(continued from previous page)

```
    //

    return ret;
}

inline
Eigen::MatrixXd
zeros_test_jacob_2(const Eigen::VectorXd& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    Eigen::MatrixXd ret(2,2);

    ret(0,0) = 2 + std::exp(-x_1);
    ret(0,1) = - 1.0;
    ret(1,0) = - 1.0;
    ret(1,1) = 2 + std::exp(-x_2);

    //

    return ret;
}

int main()
{
    Eigen::VectorXd x = Eigen::VectorXd::Zero(2); // initial values (0,0)

    bool success = optim::broyden(x, zeros_test_objfn_2, nullptr);

    if (success) {
        std::cout << "broyden: test_2 completed successfully." << "\n";
    } else {
        std::cout << "broyden: test_2 completed unsuccessfully." << "\n";
    }

    std::cout << "broyden: solution to test_2:\n" << x << std::endl;

    //

    x = Eigen::VectorXd::Zero(2);

    success = optim::broyden(x, zeros_test_objfn_2, nullptr, zeros_test_jacob_2,␣
↪nullptr);

    if (success) {
        std::cout << "broyden with jacobian: test_2 completed successfully." << "\n";
    } else {
        std::cout << "broyden with jacobian: test_2 completed unsuccessfully." << "\n";
    }
```

(continues on next page)

(continued from previous page)

```
    std::cout << "broyden with jacobian: solution to test_2:\n" << x << std::endl;

    //

    return 0;
}
```

broyden	Broyden's method for solving systems of nonlinear equations
broyden	Broyden's method for solving systems of nonlinear equations
broyden	Broyden's method for solving systems of nonlinear equations
broyden	Broyden's method for solving systems of nonlinear equations

3.9.2 Derivative-free Broyden

Table of contents

Algorithm Description

Li and Fukushima (2000) is a derivative-free variant of Broyden's method for solving systems of nonlinear equations.

We seek a vector of values $x^{(*)}$ such that

$$F(x^{(*)}) = \mathbf{0}$$

where $F : \mathbb{R}^n \to \mathbb{R}^m$ is convex and differentiable. The algorithm uses an approximation to the Jacobian.

The updating rule for Broyden's method is described below. Let $x^{(i)}$ denote the function input values at stage i of the algorithm.

1. Compute the descent direction using:

$$d^{(i)} = -B^{(i)} F(x^{(i)})$$

2. Quasi line search step.

If

$$\|F(x^{(i)} + d^{(i)})\| \leq \rho\|F(x^{(i)}\| - \sigma_2\|d^{(i)}\|^2$$

then set $\lambda_i = 1.0$; otherwise set $\eta_i = 1.0/i^2$ and find the smallest k for which

$$\|F(x^{(i)} + \lambda_i d^{(i)})\| \leq \|F(x^{(i)}\| - \sigma_1\|d^{(i)}\|^2 + \eta_i\|F(x^{(i)})\|$$

holds with $\lambda_i = \beta^k$.

3. Update the candidate solution vector using:

$$x^{(i+1)} = x^{(i)} + \lambda_i d^{(i)}$$

4. Update the approximate inverse Jacobian matrix, B, using:

$$B^{(i+1)} = B^{(i)} + \frac{1}{[y^{(i+1)}]^\top y^{(i+1)}}(s^{(i+1)} - B^{(i)}y^{(i+1)})[y^{(i+1)}]^\top$$

where

$$s^{(i)} := x^{(i)} - x^{(i-1)}$$
$$y^{(i)} := F(x^{(i)}) - F(x^{(i-1)})$$

The algorithm stops when at least one of the following conditions are met:

1. $\|F\|$ is less than `rel_objfn_change_tol`.
2. the relative change between $x^{(i+1)}$ and $x^{(i)}$ is less than `rel_sol_change_tol`;
3. the total number of iterations exceeds `iter_max`.

Function Declarations

bool **broyden_df**(ColVec_t &init_out_vals, std::function<ColVec_t(const ColVec_t &vals_inp, void *opt_data)> opt_objfn, void *opt_data)

Derivative-free variant of Broyden's method due to Li and Fukushima (2000)

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **broyden_df**(ColVec_t &init_out_vals, std::function<ColVec_t(const ColVec_t &vals_inp, void *opt_data)> opt_objfn, void *opt_data, algo_settings_t &settings)

Derivative-free variant of Broyden's method due to Li and Fukushima (2000)

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **settings** – parameters controlling the optimization routine.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **broyden_df**(ColVec_t &init_out_vals, std::function<ColVec_t(const ColVec_t &vals_inp, void *opt_data)> opt_objfn, void *opt_data, std::function<Mat_t(const ColVec_t &vals_inp, void *jacob_data)> jacob_objfn, void *jacob_data)

Derivative-free variant of Broyden's method due to Li and Fukushima (2000)

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `opt_data` additional data passed to the user-provided function.
- **opt_data** – additional data passed to the user-provided function.
- **jacob_objfn** – a function to calculate the Jacobian matrix, taking two arguments:
  - `vals_inp` a vector of inputs; and
  - `jacob_data` additional data passed to the Jacobian function.
- **jacob_data** – additional data passed to the Jacobian function.

Returns

a boolean value indicating successful completion of the optimization algorithm.

bool **broyden_df**(ColVec_t &init_out_vals, std::function<ColVec_t(const ColVec_t &vals_inp, void *opt_data)> opt_objfn, void *opt_data, std::function<Mat_t(const ColVec_t &vals_inp, void *jacob_data)> jacob_objfn, void *jacob_data, algo_settings_t &settings)

Derivative-free variant of Broyden's method due to Li and Fukushima (2000)

Parameters

- **init_out_vals** – a column vector of initial values, which will be replaced by the solution upon successful completion of the optimization algorithm.
- **opt_objfn** – the function to be minimized, taking three arguments:
  - `vals_inp` a vector of inputs; and
  - `opt_data` additional data passed to the user-provided function.

- **opt_data** – additional data passed to the user-provided function.
- **jacob_objfn** – a function to calculate the Jacobian matrix, taking two arguments:
  - `vals_inp` a vector of inputs; and
  - `jacob_data` additional data passed to the Jacobian function.
- **jacob_data** – additional data passed to the Jacobian function.
- **settings** – parameters controlling the optimization routine.

Returns
a boolean value indicating successful completion of the optimization algorithm.

Optimization Control Parameters

The basic control parameters are:

- `fp_t rel_objfn_change_tol`: the error tolerance value controlling how small $\|F\|$ should be before 'convergence' is declared.
- `fp_t rel_sol_change_tol`: the error tolerance value controlling how small the proportional change in the solution vector should be before 'convergence' is declared.

  The relative change is computed using:

$$\left\| \frac{x^{(i)} - x^{(i-1)}}{|x^{(i-1)}| + \epsilon} \right\|_1$$

  where ϵ is a small number added for numerical stability.
- `size_t iter_max`: the maximum number of iterations/updates before the algorithm exits.
- `bool vals_bound`: whether the search space of the algorithm is bounded. If `true`, then
  - `ColVec_t lower_bounds`: defines the lower bounds of the search space.
  - `ColVec_t upper_bounds`: defines the upper bounds of the search space.

In addition to these:

- `int print_level`: Set the level of detail for printing updates on optimization progress.
  - Level `0`: Nothing (default).
  - Level `1`: Print the current iteration count and error values.
  - Level `2`: Level 1 plus the current candidate solution values, $x^{(i+1)}$.
  - Level `3`: Level 2 plus the direction vector, $d^{(i)}$, and the function values, $F(x^{(i+1)})$.
  - Level `4`: Level 3 plus the components used to update the approximate inverse Jacobian matrix: $s^{(i+1)}$, $y^{(i+1)}$, and $B^{(i+1)}$.

Examples

Example 1

$$F(\mathbf{x}) = \begin{bmatrix} \exp(-\exp(-(x_1 + x_2))) - x_2(1 + x_1^2) \\ x_1 \cos(x_2) + x_2 \sin(x_1) - 0.5 \end{bmatrix}$$

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
arma::vec
zeros_test_objfn_1(const arma::vec& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    arma::vec ret(2);

    ret(0) = std::exp(-std::exp(-(x_1+x_2))) - x_2*(1 + std::pow(x_1,2));
    ret(1) = x_1*std::cos(x_2) + x_2*std::sin(x_1) - 0.5;

    //

    return ret;
}

inline
arma::mat
zeros_test_jacob_1(const arma::vec& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    arma::mat ret(2,2);

    ret(0,0) = std::exp(-std::exp(-(x_1+x_2))-(x_1+x_2)) - 2*x_1*x_1;
    ret(0,1) = std::exp(-std::exp(-(x_1+x_2))-(x_1+x_2)) - x_1*x_1 - 1.0;
    ret(1,0) = std::cos(x_2) + x_2*std::cos(x_1);
    ret(1,1) = -x_1*std::sin(x_2) + std::cos(x_1);

    //

    return ret;
```

(continues on next page)

(continued from previous page)

```
}

int main()
{
    arma::vec x = arma::zeros(2,1); // initial values (0,0)

    bool success = optim::broyden_df(x, zeros_test_objfn_1, nullptr);

    if (success) {
        std::cout << "broyden_df: test_1 completed successfully." << "\n";
    } else {
        std::cout << "broyden_df: test_1 completed unsuccessfully." << "\n";
    }

    arma::cout << "broyden_df: solution to test_1:\n" << x << arma::endl;

    //

    x = arma::zeros(2,1);

    success = optim::broyden_df(x, zeros_test_objfn_1, nullptr, zeros_test_jacob_1,␣
↪nullptr);

    if (success) {
        std::cout << "broyden_df with jacobian: test_1 completed successfully." << "\n";
    } else {
        std::cout << "broyden_df with jacobian: test_1 completed unsuccessfully." << "\n
↪";
    }

    arma::cout << "broyden_df with jacobian: solution to test_1:\n" << x << arma::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
Eigen::VectorXd
zeros_test_objfn_1(const Eigen::VectorXd& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    Eigen::VectorXd ret(2);
```

(continues on next page)

(continued from previous page)

```
    ret(0) = std::exp(-std::exp(-(x_1+x_2))) - x_2*(1 + std::pow(x_1,2));
    ret(1) = x_1*std::cos(x_2) + x_2*std::sin(x_1) - 0.5;

    //

    return ret;
}

inline
Eigen::MatrixXd
zeros_test_jacob_1(const Eigen::VectorXd& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    Eigen::MatrixXd ret(2,2);

    ret(0,0) = std::exp(-std::exp(-(x_1+x_2))-(x_1+x_2)) - 2*x_1*x_1;
    ret(0,1) = std::exp(-std::exp(-(x_1+x_2))-(x_1+x_2)) - x_1*x_1 - 1.0;
    ret(1,0) = std::cos(x_2) + x_2*std::cos(x_1);
    ret(1,1) = -x_1*std::sin(x_2) + std::cos(x_1);

    //

    return ret;
}

int main()
{
    Eigen::VectorXd x = Eigen::VectorXd::Zero(2); // initial values (0,0)

    bool success = optim::broyden_df(x, zeros_test_objfn_1, nullptr);

    if (success) {
        std::cout << "broyden_df: test_1 completed successfully." << "\n";
    } else {
        std::cout << "broyden_df: test_1 completed unsuccessfully." << "\n";
    }

    std::cout << "broyden_df: solution to test_1:\n" << x << std::endl;

    //

    x = Eigen::VectorXd::Zero(2);

    success = optim::broyden_df(x, zeros_test_objfn_1, nullptr, zeros_test_jacob_1,␣
↪nullptr);

    if (success) {
```

(continues on next page)

(continued from previous page)

```
        std::cout << "broyden_df with jacobian: test_1 completed successfully." << "\n";
    } else {
        std::cout << "broyden_df with jacobian: test_1 completed unsuccessfully." << "\n
↪";
    }

    std::cout << "broyden_df with jacobian: solution to test_1:\n" << x << std::endl;

    //

    return 0;
}
```

Example 2

$$F(\mathbf{x}) = \begin{bmatrix} 2x_1 - x_2 - \exp(-x_1) \\ -x_1 + 2x_2 - \exp(-x_2) \end{bmatrix}$$

Code to run this example is given below.

Armadillo (Click to show/hide)

```
#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "optim.hpp"

inline
arma::vec
zeros_test_objfn_2(const arma::vec& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    arma::vec ret(2);

    ret(0) =   2*x_1 - x_2   - std::exp(-x_1);
    ret(1) = - x_1   + 2*x_2 - std::exp(-x_2);

    //

    return ret;
}

inline
arma::mat
zeros_test_jacob_2(const arma::vec& vals_inp, void* opt_data)
{
```

(continues on next page)

(continued from previous page)

```
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    arma::mat ret(2,2);

    ret(0,0) = 2 + std::exp(-x_1);
    ret(0,1) = - 1.0;
    ret(1,0) = - 1.0;
    ret(1,1) = 2 + std::exp(-x_2);

    //

    return ret;
}

int main()
{
    arma::vec x = arma::zeros(2,1); // initial values (0,0)

    bool success = optim::broyden_df(x, zeros_test_objfn_2, nullptr);

    if (success) {
        std::cout << "broyden_df: test_2 completed successfully." << "\n";
    } else {
        std::cout << "broyden_df: test_2 completed unsuccessfully." << "\n";
    }

    arma::cout << "broyden_df: solution to test_2:\n" << x << arma::endl;

    //

    x = arma::zeros(2,1);

    success = optim::broyden_df(x, zeros_test_objfn_2, nullptr, zeros_test_jacob_2,␣
↪nullptr);

    if (success) {
        std::cout << "broyden_df with jacobian: test_2 completed successfully." << "\n";
    } else {
        std::cout << "broyden_df with jacobian: test_2 completed unsuccessfully." << "\n
↪";
    }

    arma::cout << "broyden_df with jacobian: solution to test_2:\n" << x << arma::endl;

    //

    return 0;
}
```

Eigen (Click to show/hide)

```cpp
#define OPTIM_ENABLE_EIGEN_WRAPPERS
#include "optim.hpp"

inline
Eigen::VectorXd
zeros_test_objfn_2(const Eigen::VectorXd& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    Eigen::VectorXd ret(2);

    ret(0) =   2*x_1 - x_2   - std::exp(-x_1);
    ret(1) = - x_1   + 2*x_2 - std::exp(-x_2);

    //

    return ret;
}

inline
Eigen::MatrixXd
zeros_test_jacob_2(const Eigen::VectorXd& vals_inp, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    //

    Eigen::MatrixXd ret(2,2);

    ret(0,0) = 2 + std::exp(-x_1);
    ret(0,1) = - 1.0;
    ret(1,0) = - 1.0;
    ret(1,1) = 2 + std::exp(-x_2);

    //

    return ret;
}

int main()
{
    Eigen::VectorXd x = Eigen::VectorXd::Zero(2); // initial values (0,0)

    bool success = optim::broyden_df(x, zeros_test_objfn_2, nullptr);

    if (success) {
        std::cout << "broyden_df: test_2 completed successfully." << "\n";
    } else {
        std::cout << "broyden_df: test_2 completed unsuccessfully." << "\n";
```

(continues on next page)

(continued from previous page)

```
    }

    std::cout << "broyden_df: solution to test_2:\n" << x << std::endl;

    //

    x = Eigen::VectorXd::Zero(2);

    success = optim::broyden_df(x, zeros_test_objfn_2, nullptr, zeros_test_jacob_2,␣
↪nullptr);

    if (success) {
        std::cout << "broyden_df with jacobian: test_2 completed successfully." << "\n";
    } else {
        std::cout << "broyden_df with jacobian: test_2 completed unsuccessfully." << "\n
↪";
    }

    std::cout << "broyden_df with jacobian: solution to test_2:\n" << x << std::endl;

    //

    return 0;
}
```

broyden-df	Derivative-free variant of Broyden's method
broyden-df	Derivative-free variant of Broyden's method
broyden-df	Derivative-free variant of Broyden's method
broyden-df	Derivative-free variant of Broyden's method

3.10 Box Constraints

This section provides implementation details for how OptimLib handles box constraints.

The problem is to transform

$$\min_{x \in X} f(x)$$

where X is a subset of $\mathbb{R}^d$, to

$$\min_{y \in \mathbb{R}^d} f(g^{-1}(y))$$

using a smooth, invertible mapping $g : X \to \mathbb{R}^d$.

OptimLib allows the user to specify upper and lower bounds for each element of the input vector, $x_j \in [a_j, b_j]$, and uses the following specification for g:

$$g(x_j) = \ln\left(\frac{x_j - a_j}{b_j - x_j}\right)$$

with corresponding inverse:

$$g^{-1}(y_j) = \frac{a_j + b_j \exp(y_j)}{1 + \exp(y_j)}$$

The gradient vector is then:

$$\nabla_y f(g^{-1}(y)) = J(y)[\nabla_{x=g^{-1}(y)} f]$$

where $J(y)$ is a $d \times d$ diagonal matrix with typical element:

$$J_{j,j} = \frac{d}{dy_j} g^{-1}(y_j) = \frac{\exp(y_j)(b_j - a_j)}{(1 + \exp(y_j))^2}$$

3.11 Line Search

For effective line search in convex optimization problems, OptimLib uses the method of More and Thuente (1994)

3.12 Test Functions

Table of contents

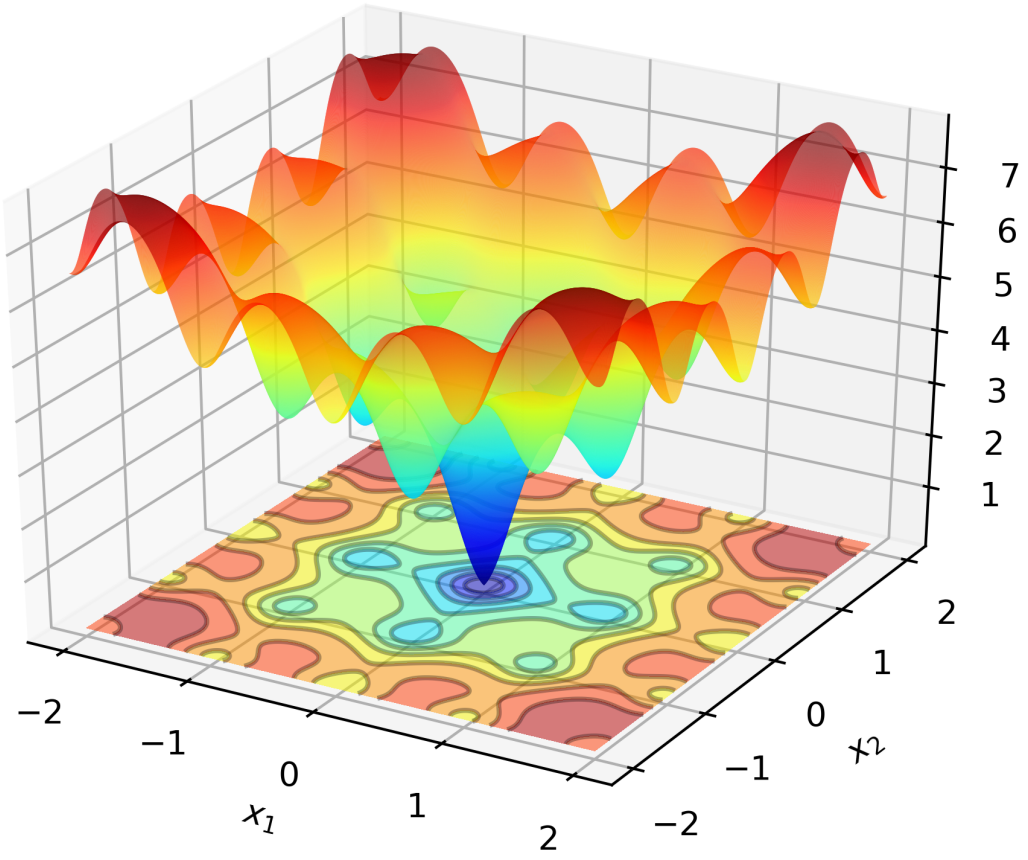
3.12.1 Ackley Function

The Ackley function is given by:

$$\min_{x \in [-5,5]^2} \left\{ 20 + \exp(1) - 20 \exp\left(-0.2 \sqrt{0.5(x_1^2 + x_2^2)} \right) - \exp\left(0.5[\cos(2\pi x_1) + \cos(2\pi x_2)]\right) \right\}$$

The minimum value is attained at $(0, 0)$.

A plot of the function is given below.



Code to run this example is given below.

Code (Click to show/hide)

```
#define OPTIM_PI 3.14159265358979

double
ackley_fn(const ColVec_t& vals_inp, ColVec_t* grad_out, void* opt_data)
{
    const double x = vals_inp(0);
    const double y = vals_inp(1);

    double obj_val = 20 + std::exp(1) - 20*std::exp( -0.2*std::sqrt(0.5*(x*x + y*y)) ) -␣
↪std::exp( 0.5*(std::cos(2 * OPTIM_PI * x) + std::cos(2 * OPTIM_PI * y)) );

    return obj_val;
}
```

3.12.2 Beale Function

The Beale function is given by:

$$\min_{x\in[-4.5,4.5]^2} \left\{(1.5 - x_1 + x_1x_2)^2 + (2.25 - x_1 + x_1x_2^2)^2 + (2.625 - x_1 + x_1x_2^3)^2\right\}$$

The minimum value is attained at $(3, 0.5)$. The function is non-convex.

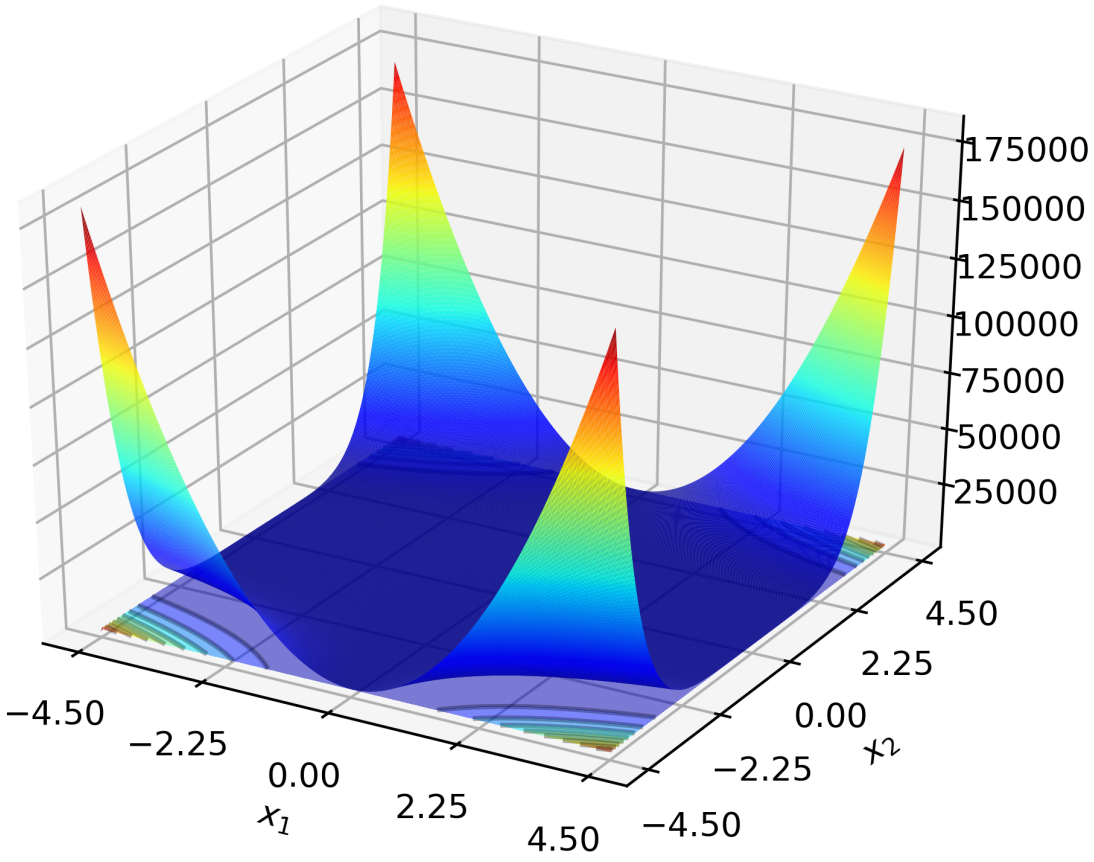
- The gradient is given by

$$\nabla_x f(x) = \begin{bmatrix} 2(1.5 - x_1 + x_1x_2)(x_2 - 1) + 2(2.25 - x_1 + x_1x_2^2)(x_2^2 - 1) + 2(2.625 - x_1 + x_1x_2^3)(x_2^3 - 1) \\ 2(1.5 - x_1 + x_1x_2)(x_1) + 2(2.25 - x_1 + x_1x_2^2)(2x_1x_2) + 2(2.625 - x_1 + x_1x_2^3)(3x_1x_2^2) \end{bmatrix}$$

- The elements of the Hessian matrix are;

$$\begin{aligned} H_{11} &= 2x_2^6 + 2x_2^4 - 4x_2^3 - 2x_2^2 - 4x_2 + 6 \\ H_{12} = H_{21} &= 12x_1x_2^5 + 8x_1x_2^3 - 12x_1x_2^2 - 4x_1x_2 - 4x_1 + 15.75x_2^2 + 9x_2 + 3 \\ H_{22} &= 30x_1^2x_2^4 + 12x_1^2x_2^2 - 12x_1^2x_2 - 2x_1^2 + 31.5x_1x_2 + 9x_1 \end{aligned}$$

A plot of the function is given below.



Code to run this example is given below.

Code (Click to show/hide)

```
double
beale_fn(const ColVec_t& vals_inp, ColVec_t* grad_out, void* opt_data)
{
    const double x_1 = vals_inp(0);
    const double x_2 = vals_inp(1);

    // compute some terms only once

    const double x2sq = x_2 * x_2;
    const double x2cb = x2sq * x_2;
    const double x1x2 = x_1*x_2;

    //

    double obj_val = std::pow(1.5 - x_1 + x1x2, 2) + std::pow(2.25 - x_1 + x_1*x2sq, 2)␣
↪+ std::pow(2.625 - x_1 + x_1*x2cb, 2);

    if (grad_out) {
        (*grad_out)(0) = 2 * ( (1.5 - x_1 + x1x2)*(x_2 - 1) + (2.25 - x_1 + x_
↪1*x2sq)*(x2sq - 1) + (2.625 - x_1 + x_1*x2cb)*(x2cb - 1)   );
        (*grad_out)(1) = 2 * ( (1.5 - x_1 + x1x2)*(x_1)     + (2.25 - x_1 + x_
↪1*x2sq)*(2*x1x2)   + (2.625 - x_1 + x_1*x2cb)*(3*x_1*x2sq) );
    }

    return obj_val;
}
```

3.12.3 Booth Function

The Booth function is given by:

$$\min_{x \in [-10,10]^2} \left\{ (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2 \right\}$$

The minimum value is attained at $(1, 3)$.

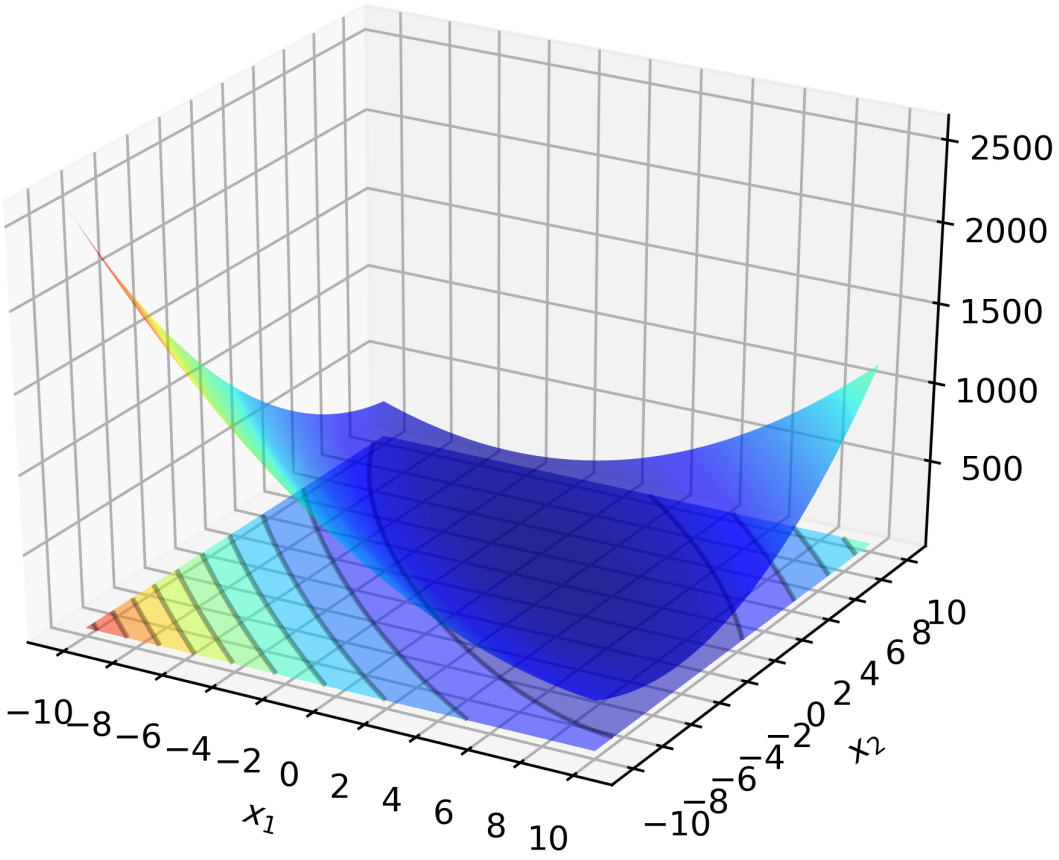
- The gradient (ignoring the box constraints) is given by

$$\nabla_x f(x) = \begin{bmatrix} 2(x_1 + 2x_2 - 7) + 4(2x_1 + x_2 - 5) \\ 4(x_1 + 2x_2 - 7) + 2(2x_1 + x_2 - 5) \end{bmatrix} = \begin{bmatrix} 10x_1 + 8x_2 - 34 \\ 8x_1 + 10x_2 - 38 \end{bmatrix}$$

- The hessian is given by

$$\nabla_x [\nabla_x^\top f(x)] = \begin{bmatrix} 10 & 8 \\ 8 & 10 \end{bmatrix}$$

A plot of the function is given below.



Code to run this example is given below.

Code (Click to show/hide)

```
double
booth_fn(const ColVec_t& vals_inp, ColVec_t* grad_out, void* opt_data)
{
    double x_1 = vals_inp(0);
    double x_2 = vals_inp(1);

    double obj_val = std::pow(x_1 + 2*x_2 - 7.0, 2) + std::pow(2*x_1 + x_2 - 5.0, 2);

    if (grad_out) {
        (*grad_out)(0) = 10*x_1 + 8*x_2 - 34;
        (*grad_out)(1) = 8*x_1 + 10*x_2 - 38;
    }

    return obj_val;
}
```

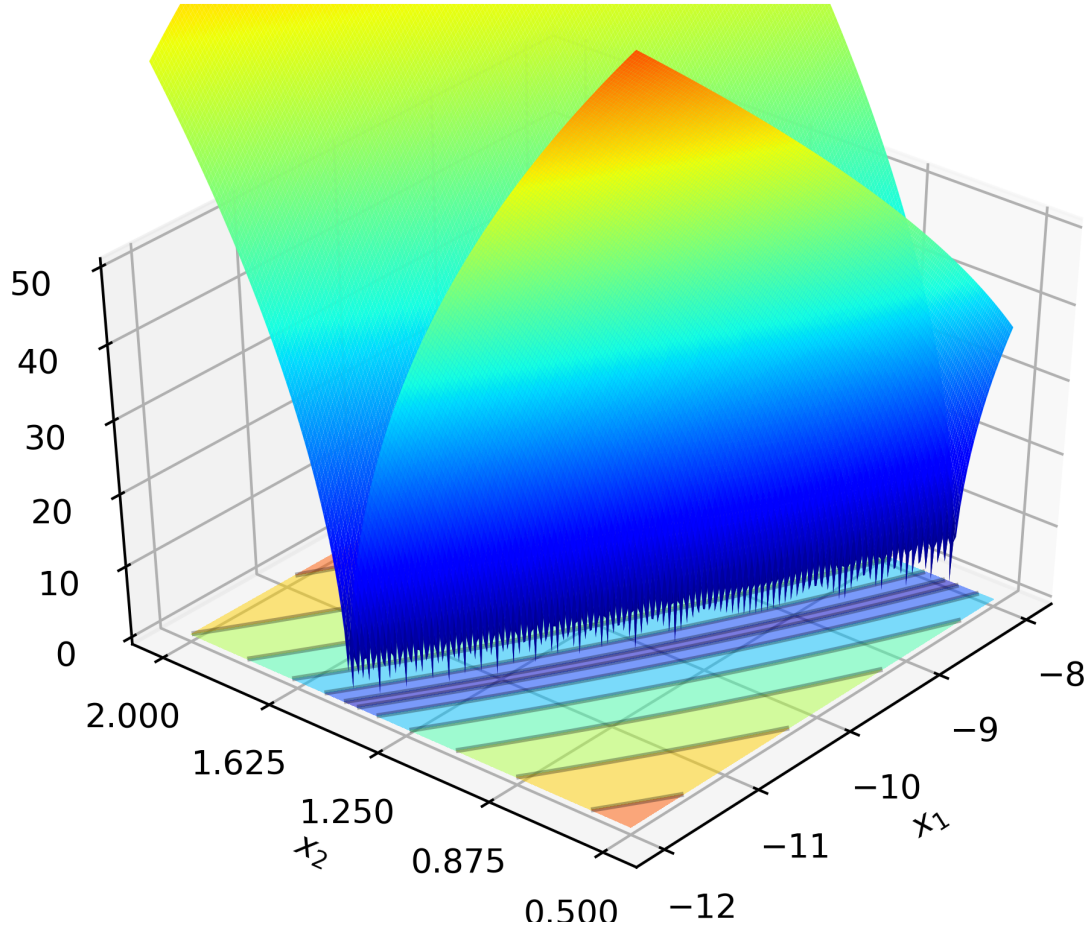
3.12.4 Bukin Function

The Bukin function (N. 6) is given by:

$$\min_{x \in [-15,-5] \times [-3,3]} \left\{ 100 \sqrt{|x_2 - 0.01 x_1^2|} + 0.01 \, |x_1 + 10| \right\}$$

The minimum value is attained at $(-10, 1)$.

A plot of the function is given below.



Code to run this example is given below.

Code (Click to show/hide)

```
double
bukin_fn(const ColVec_t& vals_inp, ColVec_t* grad_out, void* opt_data)
{
    const double x = vals_inp(0);
    const double y = vals_inp(1);

    double obj_val = 100*std::sqrt(std::abs(y - 0.01*x*x)) + 0.01*std::abs(x + 10);

    return obj_val;
}
```

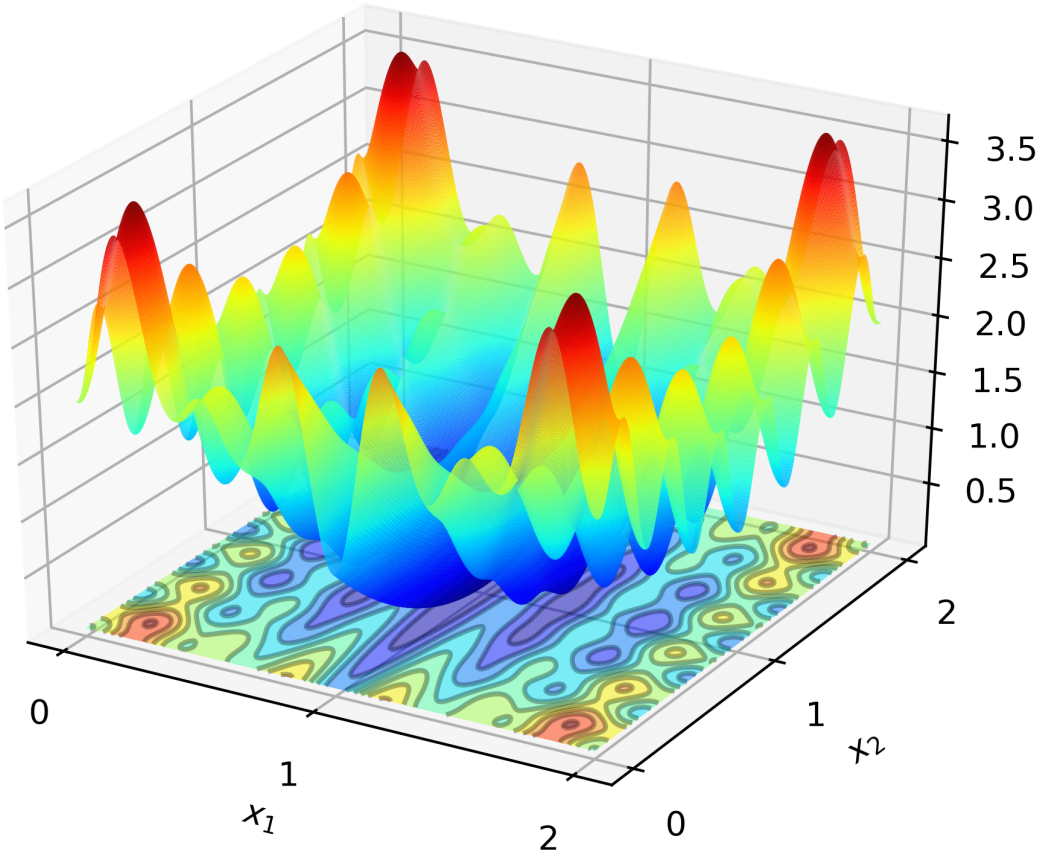
3.12.5 Levi Function

The Levi function (N. 13) is given by:

$$\min_{x \in [-10,10]^2} \left\{ (\sin(3\pi x_1))^2 + (x_1 - 1)^2 (1 + (\sin(3\pi x_2))^2) + (x_2 - 1)^2 (1 + (\sin(2\pi x_1))^2) \right\}$$

The minimum value is attained at $(1, 1)$.

A plot of the function is given below.



Code to run this example is given below.

Code (Click to show/hide)

```
#define OPTIM_PI 3.14159265358979

double
levi_fn(const ColVec_t& vals_inp, ColVec_t* grad_out, void* opt_data)
{
    const double x = vals_inp(0);
    const double y = vals_inp(1);
    const double pi = OPTIM_PI;

    double obj_val = std::pow( std::sin(3*pi*x), 2) + std::pow(x-1,2)*(1 + std::pow(␣
↪std::sin(3 * OPTIM_PI * y), 2)) + std::pow(y-1,2)*(1 + std::pow( std::sin(2 * OPTIM_PI␣
```

(continues on next page)

(continued from previous page)

```
↪* x), 2));

    return obj_val;
}
```

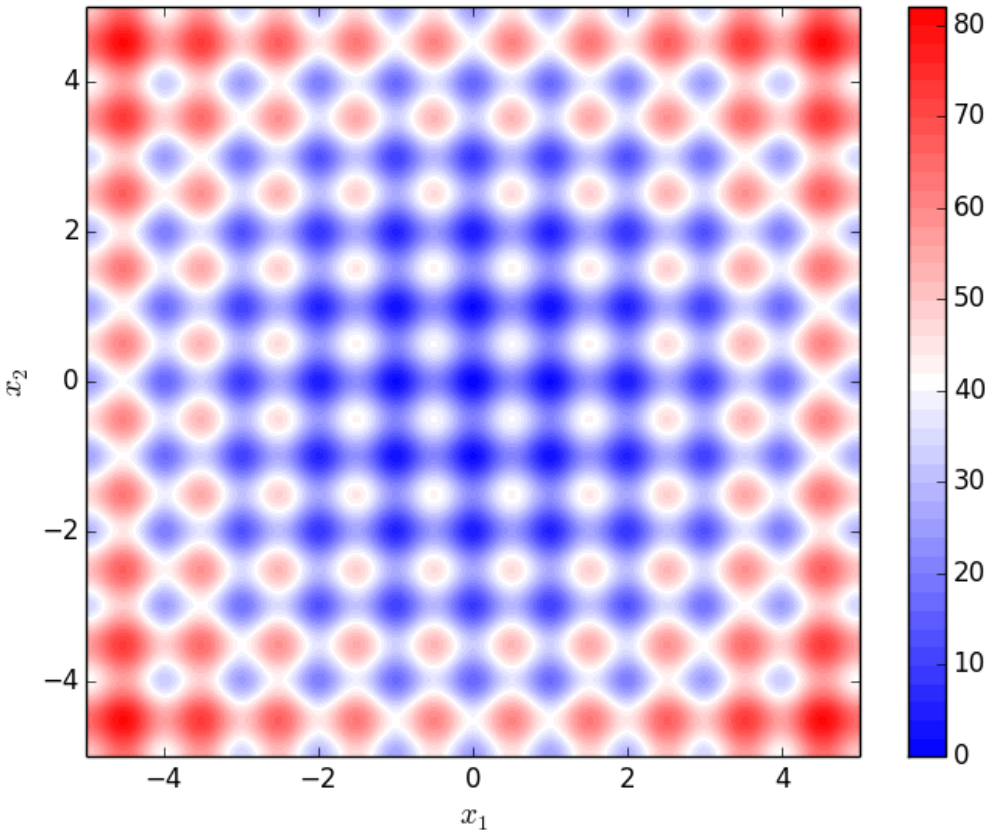
3.12.6 Rastrigin Function

The 2D Rastrigin function is given by:

$$\min_{x \in [-5,5]} \left\{ 20 + (x_1^2 - 10 \times \cos(2\pi x_1)) + (x_2^2 - 10 \times \cos(2\pi x_2)) \right\}$$

The minimum value is attained at $(0, 0)$.

A plot of the function is given below.



Code to run this example is given below.

Code (Click to show/hide)

```
#define OPTIM_PI 3.14159265358979

double
rastrigin_fn(const ColVec_t& vals_inp, ColVec_t* grad_out, void* opt_data)
{
    const double x_1 = vals_inp(0);
    const double x_2 = vals_inp(1);

    double obj_val = 20 + x_1*x_1 + x_2*x_2 - 10 * (std::cos(2*OPTIM_PI*x_1) +␣
↪std::cos(2*OPTIM_PI*x_2))

    return obj_val;
}
```

3.12.7 Rosenbrock Function

The 2D Rosenbrock function is given by:

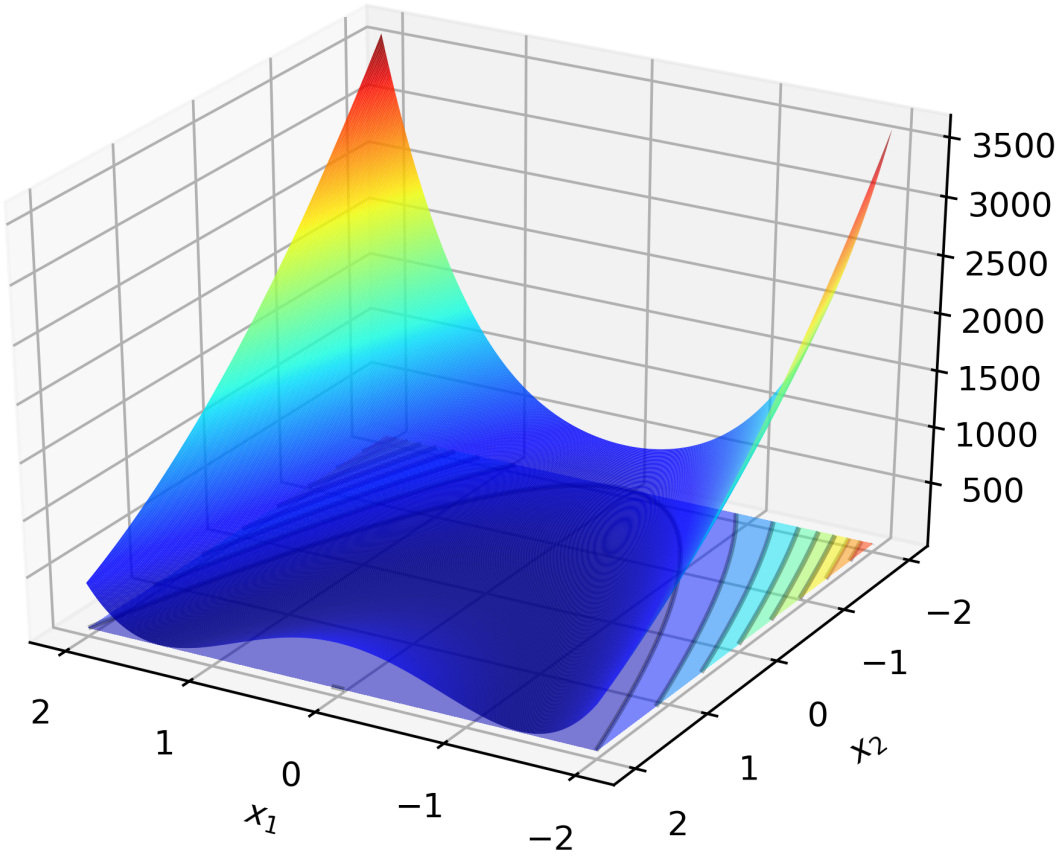
$$\min_{x \in \mathbb{R}^2} \left\{ 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \right\}$$

The minimum value is attained at $(1, 1)$.

- The gradient is given by

$$\nabla_x f(x) = \begin{bmatrix} -400(x_2 - x_1^2)(x_1) - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{bmatrix}$$

A plot of the function is given below.



Code to run this example is given below.

Code (Click to show/hide)

```
double
rosenbrock_fn(const ColVec_t& vals_inp, ColVec_t* grad_out, void* opt_data)
{
    const double x_1 = vals_inp(0);
    const double x_2 = vals_inp(1);

    const double x1sq = x_1 * x_1;

    double obj_val = 100*std::pow(x_2 - x1sq,2) + std::pow(1-x_1,2);

    if (grad_out) {
        (*grad_out)(0) = -400*(x_2 - x1sq)*x_1 - 2*(1-x_1);
        (*grad_out)(1) = 200*(x_2 - x1sq);
    }

    return obj_val;
}
```

3.12.8 Sphere Function

The Sphere function is a very simple smooth test function, given by:

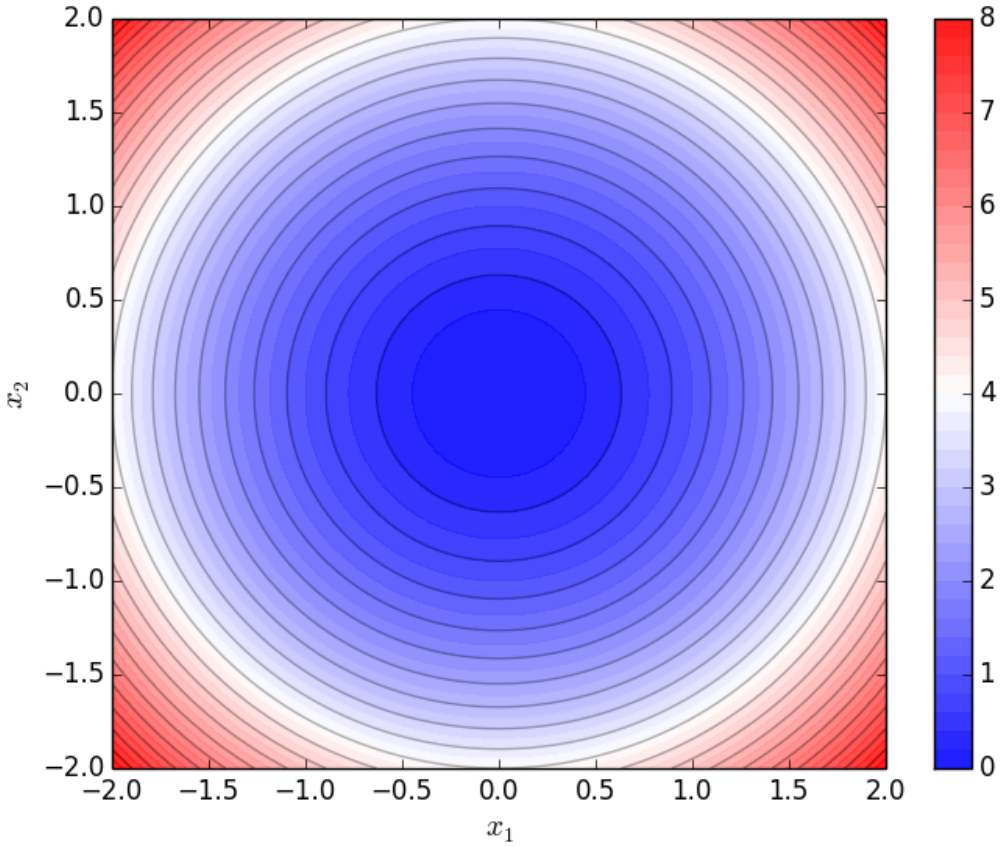
$$\min_{x \in \mathbb{R}^n} \left\{ \sum_{i=1}^{n} x_i^2 \right\}$$

The minimum value is attained at the origin.

- The gradient is given by

$$\nabla_x f(x) = [2x_1, \ldots, 2x_n]^\top$$

A contour plot of the Sphere function in two dimensions is given below.



Code to run this example is given below.

Armadillo (Click to show/hide)

```
#include "optim.hpp"

inline
double
sphere_fn(const arma::vec& vals_inp, arma::vec* grad_out, void* opt_data)
{
    double obj_val = arma::dot(vals_inp,vals_inp);
```

(continues on next page)

(continued from previous page)

```
    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
    const int test_dim = 5;

    arma::vec x = arma::ones(test_dim,1); // initial values (1,1,...,1)

    bool success = optim::bfgs(x, sphere_fn, nullptr);

    if (success) {
        std::cout << "bfgs: sphere test completed successfully." << "\n";
    } else {
        std::cout << "bfgs: sphere test completed unsuccessfully." << "\n";
    }

    arma::cout << "bfgs: solution to sphere test:\n" << x << arma::endl;

    return 0;
}
```

Eigen (Click to show/hide)

```
#include "optim.hpp"

inline
double
sphere_fn(const Eigen::VectorXd& vals_inp, Eigen::VectorXd* grad_out, void* opt_data)
{
    double obj_val = vals_inp.dot(vals_inp);

    if (grad_out) {
        *grad_out = 2.0*vals_inp;
    }

    return obj_val;
}

int main()
{
    const int test_dim = 5;

    Eigen::VectorXd x = Eigen::VectorXd::Ones(test_dim); // initial values (1,1,...,1)

    bool success = optim::bfgs(x, sphere_fn, nullptr);
```

(continues on next page)

(continued from previous page)

```
    if (success) {
        std::cout << "bfgs: sphere test completed successfully." << "\n";
    } else {
        std::cout << "bfgs: sphere test completed unsuccessfully." << "\n";
    }

    std::cout << "bfgs: solution to sphere test:\n" << x << std::endl;

    return 0;
}
```

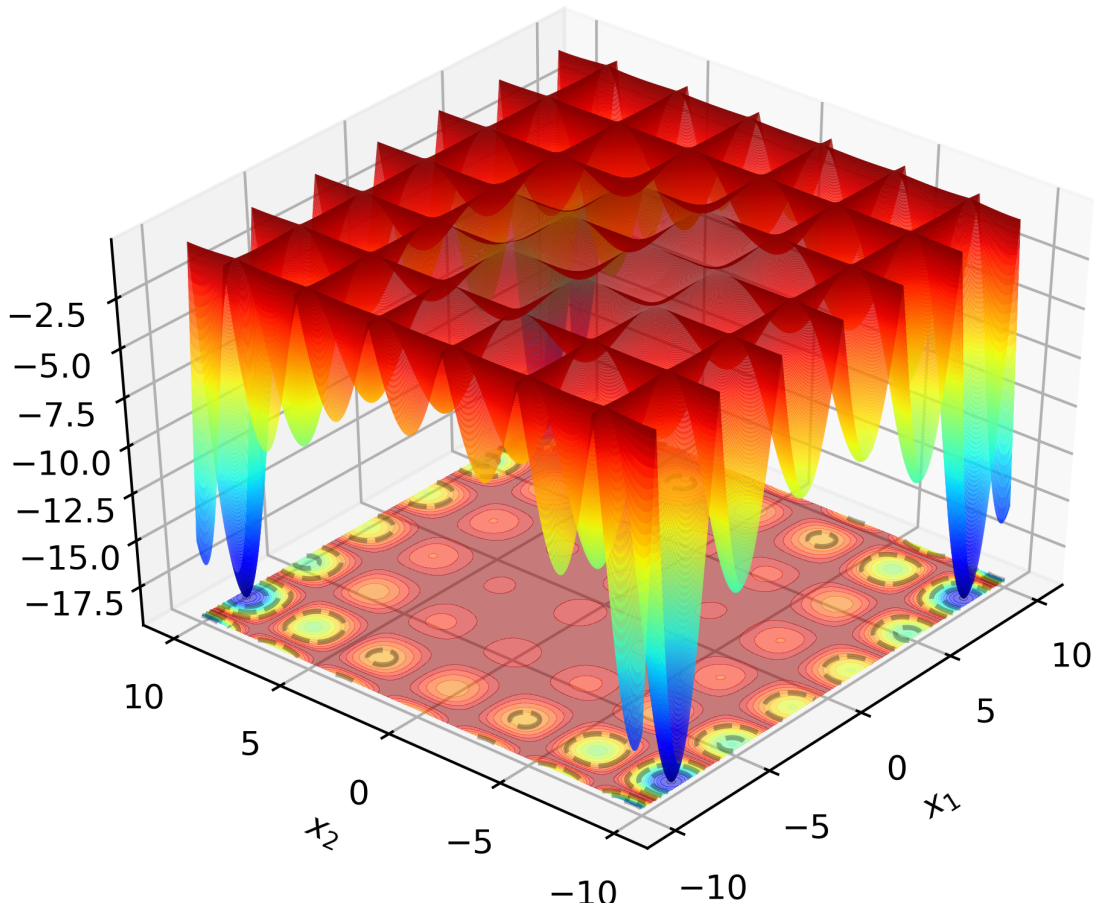
3.12.9 Table Function

The Hoelder Table function is given by:

$$\min_{x \in [-10,10]^2} \left\{ -\left| \sin(x_1)\cos(x_2)\exp\left(\left| 1 - \frac{\sqrt{x_1^2 + x_2^2}}{\pi} \right| \right) \right| \right\}$$

The minimum value is attained at four locations: $(\pm 8.05502, \pm 9.66459)$.

A plot of the function is given below.



Code to run this example is given below.

Code (Click to show/hide)

```
#define OPTIM_PI 3.14159265358979

double
table_fn(const ColVec_t& vals_inp, ColVec_t* grad_out, void* opt_data)
{
    const double x = vals_inp(0);
    const double y = vals_inp(1);

    double obj_val = - std::abs( std::sin(x)*std::cos(y)*std::exp( std::abs( 1.0 -␣
↪std::sqrt(x*x + y*y) / OPTIM_PI ) ) );

    return obj_val;
}
```

INDEX