1 Chapter 3

1.1 Exercise 1

The Rosenbrock function

$$f : \mathbb{R}^2 \to \mathbb{R}, x \mapsto 100(x_2 - x_1^2)^2 + (1 - x_1)^2,$$

also compare `http://en.wikipedia.org/wiki/Rosenbrock_function`, is frequently utilized to test optimization methods.

a. The absolute minimum of f, at the points $(1, 1)^T$, can be seen without any calculations. Show that it is the only extremal point.

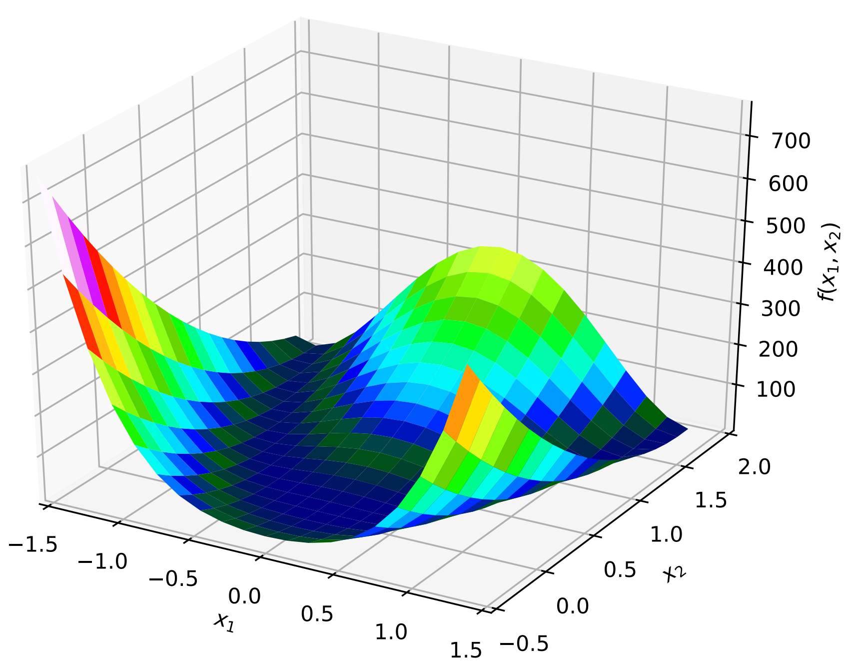
   All local (and thus global) extrema occur at critical points of f, therefore it is sufficient to show that the only critical point exists at $(1, 1)^T$.

$$\nabla f(x) = \begin{pmatrix} 200(x_2 - x_x^2)(-2x_1) - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{pmatrix}$$

   The only solution to $\nabla f(x) = 0$ is $x = (1, 1)^T$, which is at the absolute minimum $(1, 1)^T$, which we already have. There are no other critical points, therefore this is the only extrema.

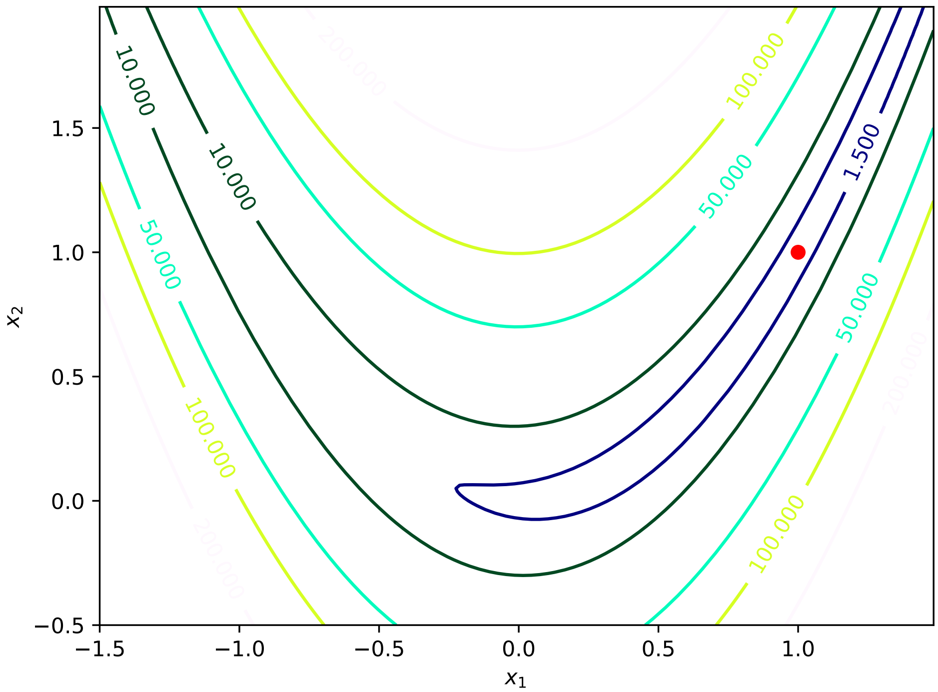
b. Graph the function f on $[-1.5, 1.5] \times [-0.5, 2]$ as a 3D plot or as a level curve plot to understand why f is referred to as the banana function and the like.

Figure 1: The Rosenbrock Function



c. Implement the Nelder-Mead method. Visualize the level curves of the given function together with the polytope for each iteration. Finally visualize the trajectory of the centres of gravity of the polytopes!

Figure 2: Convergence of polytopes to solution



d. Test the program with the starting polytope given by the vertices $(-1,1)^T$, $(0,1)^T$, $(-0.5,2)^T$ and the parameters $(\alpha,\beta,\gamma) := (1,2,0.5)$ and $\varepsilon := 10^{-4}$ using the Rosenbrock function. How many iterations are needed? What is the distance between the calculated solution and the exact minimizer $(1,1)^T$?

e. Find $(\alpha,\beta,\gamma) \in [0.9,1.1]\times[1.9,2.1]\times[0.4,0.6]$ such that with $\varepsilon := 10^{-4}$ the algorithm terminates after as few iterations as possible. What is the distance between the solution and the minimizer $(1,1)^T$ in this case?

f. If the distance in e was greater than in d, reduce ε until the algorithm gives a result—with the (α,β,γ), found in e—which is not farther away from $(1,1)^T$ and at the same time needs fewer iterations than the solution in d.

2 Appendix I

```
#pragma once

#include <algorithm>
#include <array>
#include <functional>
#include <iostream>

#include <CoreMath.hpp>
#include <Vec.hpp>

using std::array;
using std::cout;
using std::endl;
using std::function;

namespace arc {

    typedef array<Vec2d,3> Simplex2d;
    //using Simplex2d = array<Vec2d, 3>;

    /**
     *
     * @brief Minimizes function using Nelder-Mead polytope algorithm.
     * @brief Iterates f
     * @param obj Objective function to minimize
     * @param initialSimplex Starting polytop
     * @param tol Desired tolerance for solution
     * @param maxIterations Maximum number of iterations
     * @param alpha Reflection coefficient $\alpha$
     * @param beta Reflection coefficient $\beta$
     * @param gamma Contraction coefficient $\gamma$
```

```
     * @param verbose Enable verbose debug info
     * @return Solution to minimization of objective function
     *
     **/
    Vec2d minimizeNelderMead(const function<double(Vec2d)> &obj,
                             Simplex2d initialSimplex, double tol,
                             size_t maxIterations, double aplha, double beta, double gamma, bool verbose);
}
```

```
#include "MinNelderMead.hpp"

using std::stable_sort;

namespace arc {

    // -- minimizeNelderMead function --
    Vec2d minimizeNelderMead(const function<double(Vec2d)> &obj,
                             Simplex2d initialSimplex, double tol,
                             size_t maxIterations, double alpha, double beta,
                             double gamma, bool verbose = false) {
        assert(tol > 0.0);
        assert(maxIterations > 0);

        cout << "Running Nelder-Mead minimzation ..." << endl;
        cout << "------------------------------------" << endl;
        cout << "α=" << alpha << endl;
        cout << "β=" << beta << endl;
        cout << "γ=" << gamma << endl;
        cout << "with max iterations N=" << maxIterations << endl;
        cout << "to tolerance TOL ε=" << tol << endl;
        cout << "------------------------------------" << endl;
        cout << endl;

        struct ObjAtX {
            Vec2d x;
            double f;
            ObjAtX() = default;
            ObjAtX(Vec2d const &x, double f) {
                this->x = x;
                this->f = f;
            }
        };

        auto n = maxIterations;

        auto x = initialSimplex;
        array<double, 3> f{{obj(x[0]), obj(x[1]), obj(x[2])}};
        array<ObjAtX, 3> objAtX;
        objAtX[0] = {x[0], f[0]};
        objAtX[1] = {x[1], f[1]};
        objAtX[2] = {x[2], f[2]};

        // auto printState = [](array<ObjAtX, 3> const &state) -> void {
        // for (size_t i = 0; i < state.size(); i++) {
        // cout << i << ": x=" << state[i].x << ", f=";
        // cout << state[i].f << endl;
        //}
        //};

        // Parameter 1: Reflection coefficient
        // double alpha = 1.0;
        // Parameter 2: Expansion coefficient
        // double beta = 2.0;
        // Parameter 3: Contraction coefficient
        // double gamma = 0.5;

        // Sort polytop
        stable_sort(objAtX.begin(), objAtX.end(),
                    [](const ObjAtX &a, const ObjAtX &b) -> bool {
                        return a.f <= b.f;
                    });

        // Main algorithm iteration loop
        for (int k = 0; k < (int)n; k++) {
            // Set some reference variables to simply notation
            auto &x0 = objAtX[0].x;
            auto &x1 = objAtX[1].x;
            auto &x2 = objAtX[2].x;
            auto &f0 = objAtX[0].f;
            auto &f1 = objAtX[1].f;
            auto &f2 = objAtX[2].f;

            // Calculate midpoint of best simplex side
            auto xC = 0.5 * (x0 + x1);

            // Calculate reflection across best side
            auto xR = xC + alpha * (xC - x2);
            auto fR = obj(xR);

            // Case 1: f_1 <= f^r < f_n
            if (f0 <= fR && fR < f1) {
                // cout << "case 1" << endl;
                x2 = xR;
                f2 = fR;
            }
            // Case 2: f^r < f_1
            else if (fR < f0) {
                // cout << "Case 2" << endl;
                // Calculate extrapolated point x^e
                auto xE = xC + beta * (xR - xC);
                auto fE = obj(xE);
```

```
                if (fE < fR) {
                    x2 = xE;
                    f2 = fE;
                } else {
                    x2 = xR;
                    f2 = fR;
                }
            }
            // Case 3: f^r > f_n : the polytope seems too big
            else if (fR >= f1) {
                // cout << ”Case 3: Polytope too big.” << endl;
                if (fR >= f2) {
                    auto xK = xC + gamma * (x2 - xC);
                    auto fK = obj(xK);
                    if (fK < f2) {
                        x2 = xK;
                        f2 = fK;
                    } else {
                        x1 = 0.5 * (x1 + x0);
                        x2 = 0.5 * (x2 + x0);
                        f1 = obj(x1);
                        f2 = obj(x2);
                        assert(x0 == 0.5 * (x0 + x0));
                        // This doesn’t do anything skip it
                        // x0 = 0.5 * (x0 + x0);
                    }
                }
                //
                else if (fR < f2) {
                    auto xK = xC + gamma * (xR - xC);
                    auto fK = obj(xK);
                    if (fK <= fR) {
                        x2 = xK;
                        f2 = fK;
                    } else {
                        x1 = 0.5 * (x1 + x0);
                        x2 = 0.5 * (x2 + x0);
                        f1 = obj(x1);
                        f2 = obj(x2);
                    }
                }

            } else {
                assert(false);
                // cout << ”Error: No case” << endl;
            }

            // Sort polytop
            stable_sort(objAtX.begin(), objAtX.end(),
                        [](const ObjAtX &a, const ObjAtX &b) -> bool {
                            return a.f <= b.f;
                        });

            // Check termination condition
            double sum = 0.0;
            // Recalculate midpoint on best side
            xC = 0.5 * (objAtX[0].x + objAtX[1].x);
            auto fC = obj(xC);

            for (auto objX : objAtX) {
                auto diff = abs(obj(objX.x) - fC);
                diff = diff * diff;
                sum += diff;
            }

            auto measure = sum * (1.0 / 3.0);

            if (measure < tol * tol) {
                cout << ”Tolerance reached after k=” << k + 1 << ” iterations.”
                     << endl;
                auto solution = (objAtX[0].x + objAtX[1].x + objAtX[2].x) / 3.0;
                cout << ”Solution ⊠x=” << solution << ”⊠” << endl;
                cout << ”f(⊠x)=” << obj(solution);
                cout << ”⊠” << endl;

                return solution;
            }
        }

        cout << ”Error: Failed to find solution in n=” << n;
        cout << ” iterations” << endl;
        return Vec2d(0.0);
    }
}
```