1 Golden Method

a. Consider the Golden method for $\min g(x)$ in $[a, b]$. Prove that $1/R = (\sqrt{5} - 1)/2$.

Let $[a_k, b_k] \subset [a, b]$ be the reduced interval at the k^{th} step, then its length is $I_k = b_k - a_k$. The $(k+1)^{th}$ step will see the interval reduced to $[a_{k+1}, b_{k+1}]$, with length $I_{k+1} = b_{k+1} - a_{k+1}$.

To determine the ratio R between consequetive intervals I_k and I_{k+1} we require

$$\text{①} \quad \frac{I_k}{I_{k+1}} = R, \ \forall k \geq 1, \text{and}$$

$$\text{②} \quad I_k = I_{k+1} + I_{k+2}, \ \forall k \geq 1.$$

Plugging ② into ① gives

$$\frac{I_{k+1} + I_{k+2}}{I_{k+1}} = R \quad \Longrightarrow \quad 1 + R = \frac{1}{R} \quad \Longrightarrow \quad R = \frac{1+\sqrt{5}}{2} \quad \Longrightarrow \quad \frac{1}{R} = \frac{2}{1+\sqrt{5}} = \frac{\sqrt{5}-1}{2}$$

Note: R is a ratio, so the negative root has no real meaning.

b. Use the Golden Method to find the minimum point of function

$$g(x) = 3 \sin x \cos x + 2$$

in $[a_1, b_1] = [1.5, 3.2]$ with $k = 4$ (stop at $k = 4$).

c. Explain two advantages of the Golden Method over a search method with $\rho = 2/3$.

2 Fibonacci Method

a. Consider the Fibonacci method for solving $\min g(x)$ in $[a, b]$. Prove that

$$\frac{I_n}{I_{n-k}} = \frac{1}{F_{k+1}}, k = 1, \ldots, n-1$$

$$\frac{I_k}{I_1} = \frac{F_{n-k+1}}{F_n}, k = 2, 3, \ldots, n,$$

where $F_j, j > 0$ are the Fibonacci numbers.

b. Let stop be at $n = 9$ in the Fibonacci method. Find $I_2/I_1, I_4/I_1, I_8/I_1$.

3 Simplex Method

We will limit ourselves to the $2D$ version i.e. $n = 2$. I will use the following notation instead. Let $\boldsymbol{x}_1^{(k)}, \boldsymbol{x}_2^{(k)}, \boldsymbol{x}_3^{(k)}$ be the ordered vertices of the k^{th} simplex Δ_k, i.e., s.t.

$$g(\boldsymbol{x}_1^{(k)}) \le g(\boldsymbol{x}_2^{(k)}) \le g(\boldsymbol{x}_3^{(k)}),\ \forall k \ge 1.$$

a. In the Simplex method, give the formulas of locations of $a', a'', a^*, a^{**}, \bar{a}$, and $\bar{b}$.

$\alpha > 0$, $\beta > 1$, $0 < \gamma < 1$, and σ are the reflection, expansion, contraction, and shrink parameters respecitvely.

Typical values for the parameters are: $\alpha = 1, \beta = 2, \gamma = 1/2$ and $\sigma = 1/2$.

The centroid (midpoint) $\boldsymbol{x}_o^{(k)}$ is given by

$$\boldsymbol{x}_o^{(k)} = \frac{1}{2}\left(\boldsymbol{x}_1^{(k)} + \boldsymbol{x}_2^{(k)}\right)$$

The reflected point is

$$\boldsymbol{x}_r^{(k)} = \boldsymbol{x}_o^{(k)} + \alpha\left(\boldsymbol{x}_o^{(k)} - \boldsymbol{x}_3^{(k)}\right)$$

The expansion point is

$$\boldsymbol{x}_e^{(k)} = \boldsymbol{x}_o + \beta\left(\boldsymbol{x}_r^{(k)} - \boldsymbol{x}_o^{(k)}\right)$$

The contraction point is

$$\boldsymbol{x}_c^{(k)} = \boldsymbol{x}_o^{(k)} + \gamma\left((\boldsymbol{x}_3^{(k)} - \boldsymbol{x}_o^{(k)}\right)$$

The shrink vertices are given by

$$\boldsymbol{x}_i^{(k)} = \boldsymbol{x}_1^{(k)} + \sigma\left(\boldsymbol{x}_i^{(k)} - \boldsymbol{x}_1^{(k)}\right),\ \text{for } i = 2, 3$$

b. Consider the minimization function

$$g(x, y) = 2x^2 - x - 2y + y^2 + 4.$$

Starting from the initial triangle $\Delta_0 = \Delta \boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_3$, where

$$a_0 = \begin{pmatrix} 0.1 \\ 0 \end{pmatrix}, b_0 = \begin{pmatrix} 0.0 \\ 0.1 \end{pmatrix}, \text{and } c_0 = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}.$$

do two steps (i.e. find Δ_2) by the simplex method. What is your approximation to the minimum point? What are the advantages and drawbacks of the simplex method?

Operation	k	x11	x12	x21	x22	x31	x32	g1	g2	g3
`input`	0	0.0	0.1	0.1	0.0	0.0	0.0	3.81	3.92	4.0
`expand`	1	0.15	0.15	0.0	0.1	0.1	0.0	3.62	3.81	3.92
`expand`	2	0.025	0.375	0.15	0.15	0.0	0.1	3.37	3.62	3.81
`expand`	3	0.253	0.588	0.025	0.375	0.150	0.150	3.045	3.367	3.618

Obviously, after only $k = 2$ iterations, `minNelderMead` fails to reach any significant tolerance. It gives the following solution $\boldsymbol{x}^* \approx \boldsymbol{x}^{(2)} = \begin{pmatrix} 0.0583 \\ 0.2083 \end{pmatrix}$. See (Section 5) for an example of the implementation of the simplex method.

4 Steepest Descent Method

Consider the Steepest Descent method for solving the local minimization of $\min g(\boldsymbol{x})$ in $\Omega \subset \mathbb{R}^n$.

a. If the previous approximation is $\boldsymbol{x}^{(k-1)}$, what is the k'th step search direction $\boldsymbol{z}^{(k)}$? Explain briefly why you use this search direction.

b. Write out the Algorithm of the Steepest Descent Method. You need to provide the following details:

5 Application

Let

$$g(x,y) = -\left(x^2 + 4xy + 2y^2\right) e^{-2x^2 - y^2}.$$

Use a computer to approximate the local minimization problem of $\min g(x, y)$ in $\mathbb{R}^2$ by one of the numerical methods: Newton's method, the Steepest Descent method, or the simplex method.

a. Explain briefly how to solve the problem by the method that you used.

b. Set up a table of numerical results and iteration numbers by using initial guesses and tolerances. Analyze your results.

c. What are the advantages and drawbacks of the method based on your analysis?

5.1 Implementation of the simplex method

I have created a C++17 implementation of the simplex method. It is primarily implemented in the function `arc::minNelderMead`, declared in the `<MinNelderMead.hpp>` header.

The algorithm's guts can be found the in implementation file `<MinNelderMead.cpp>`. No attempt at optimization has been made. The internal state can be stored and read by passing a pointer via the `info` argument. The code is well-commented and should be self explanatory.

C++ Listing 5.2: `arc::minNelderMead` Implementation

```
#include "MinNelderMead.hpp"

using std::stable_sort;

namespace arc {

    using OpType = NelderMeadAlgoInfo::OpType;

    // -- minimizeNelderMead function --
    Vec2d minimizeNelderMead(const function<double(Vec2d)> &obj,
                             Simplex2d initialSimplex,
                             double tol,
                             size_t maxIterations,
                             double alpha,
                             double beta,
                             double gamma,
                             double sigma,
                             NelderMeadAlgoInfo *info,
                             bool verbose) {
        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;

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

        using Pair = pair<Vec2d, double>;

        array<Pair, 3> pairs = {{make_pair(initialSimplex[0], 0.0),
                                 make_pair(initialSimplex[1], 0.0),
                                 make_pair(initialSimplex[2], 0.0)}};

        auto &x0 = pairs[0].first;
        auto &x1 = pairs[1].first;
        auto &x2 = pairs[2].first;

        auto &f0 = pairs[0].second;
        auto &f1 = pairs[1].second;
        auto &f2 = pairs[2].second;

        f0 = obj(x0);
        f1 = obj(x1);
        f2 = obj(x2);

        // cout << "Before:" << endl;
        // cout << x0 << "=" << f0 << endl;
        // cout << x1 << "=" << f1 << endl;
        // cout << x2 << "=" << f2 << endl;
        // cout << endl;

        // Sort initial simplex before starting.
        stable_sort(pairs.begin(), pairs.end(),
                    [](const Pair &p0, const Pair &p1) -> bool {
                        return p0.second <= p1.second;
                    });

        // -- Main algorithm iteration loop --
        for (int k = 0; k < (int)n; k++) {
            // Output internal state info
            info->simplices.emplace_back();
            auto &s = info->simplices.back();
            s[0] = x0;
            s[1] = x1;
            s[2] = x2;

            info->fValues.emplace_back();
            info->fValues.back()[0] = f0;
            info->fValues.back()[1] = f1;
            info->fValues.back()[2] = f2;

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

            // Calculate reflection over 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) {
                x2 = xR;
                f2 = fR;

                info->opCountReflect++;
                info->ops.push_back(OpType::Reflect);
            }
            // Case 2
```

```
        else if (fR < f0) {
            // 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;
            }
            info->ops.push_back(OpType::Expand);
        }
        // Case 3: f^r > f_n : the simplex seems too big
        else if (fR >= f1) {
            info->ops.push_back(OpType::Contract);
            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 is not necessary 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);
        }

        // Sort simplex vertices.
        stable_sort(pairs.begin(), pairs.end(),
                    [](const Pair &p0, const Pair &p1) -> bool {
                        return p0.second <= p1.second;
                    });

        // Check stop condition.

        // Recalculate midpoint on best side
        xC = 0.5 * (x0 + x1);
        auto fC = obj(xC);

        auto diff0 = std::abs(f0 - fC);
        auto diff1 = std::abs(f1 - fC);
        auto diff2 = std::abs(f2 - fC);
        auto sum = diff0 * diff0 + diff1 * diff1 + diff2 * diff2;
        auto measureSq = sum / 3.0;

        if (measureSq < tol * tol) {
            cout << "done" << endl;
            auto solution = (x0 + x1 + x2) / 3.0;
            return solution;
        }
```

```
            if (k + 1 >= (int)n) {
                cout << "Warning: Failed to reach TOL in n=" << n;
                cout << " iterations!" << endl;
                auto solution = (x0 + x1 + x2) / 3.0;
                return solution;
            }
        }


        return Vec2d::zero();
    }
}  // namespace arc
```

Appendices

A Nelder-Mead function declaration and supporting objects

C++ Listing 5.1: Function declaration found in `MinNelderMead.hpp`.

```cpp
#pragma once

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

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

using std::array;
using std::cout;
using std::endl;
using std::function;
using std::make_pair;
using std::pair;
using std::vector;

namespace arc {

    typedef array<Vec2d, 3> Simplex2d;

    /**
     *
     * @brief Storage object for Nelder-Mead algorithm internal state.
     *
     **/
    struct NelderMeadAlgoInfo {
        Simplex2d simplex0;
        size_t maxN;
        double tol, alpha, beta, gamma, sigma;
        Vec2d result;
        vector<Simplex2d> simplices;
        vector<Vec2d> baryCenters;
        vector<array<double, 3>> fValues;

        size_t opCountReflect = 0;
        size_t opcountExpand = 0;
        size_t opCountContract = 0;
        size_t opCountShrink = 0;

        enum class OpType { Reflect = 0, Expand = 1, Contract = 2, Shrink = 3 };
        vector<OpType> ops;
    };

    /**
     *
     * @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,
                                double sigma,
                                NelderMeadAlgoInfo *info,
                                bool verbose = false);
}  // namespace arc
```