# Lab 16 Optimization I (Unconstrained)

1. **Gradient Estimation**

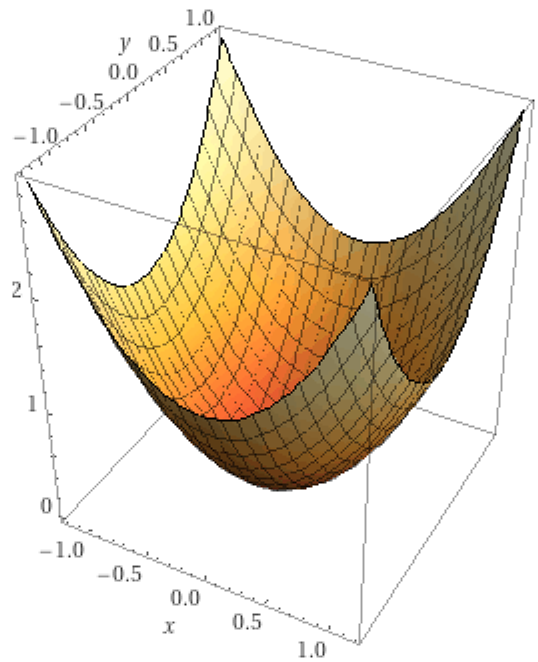
As seen in lecture, gradient descent is one of the most common ways to solve convex problems. In this lab, we will implement a numerical gradient estimation. One of the simplest variants:

- finite difference using Newton's Difference Quotient $Df(x) \approx \frac{f(x+h)-f(x)}{h}$
- use fixed `h`

First let us implement

```
#include <math.h>
#include <iostream>
#include <eigen3/Eigen/Eigen>

template <typename Function>
Eigen::VectorXd calculateGradient(Function f, Eigen::VectorXd x, double h) {
    // assumes f is a scalar function with vector input
    // thus gradient has the same dimension as the input (each is just the partial)
    // --- Your code here
    // ---
}

double sampleQuadratic(Eigen::Vector2d xy) {
    // x^2 + y^2 = [x y] [1 0; 0 1] [x y]^T
    return xy(0) * xy(0) + xy(1) * xy(1);
}

int main() {
    Eigen::Vector2d x;
    x << 0, 0;

    auto h = 0.01;
    auto d = calculateGradient(sampleQuadratic, x, h);
    std::cout << d << std::endl;
}
```

This sample quadratic function is very simple:

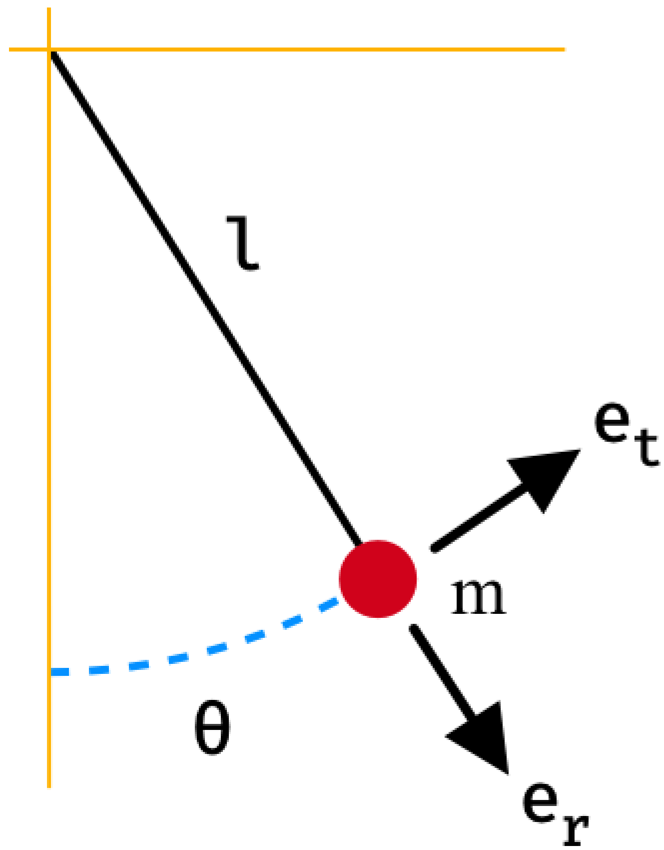



2. **Jacobian Estimation**

Gradients are for scalar functions what Jacobians are for multidimensional functions. Recall:

$$Df(x)_{ij} = \frac{\partial f_i(x)}{\partial x_j}, \qquad i = 1, \ldots, m, \quad j = 1, \ldots, n$$

Note the rows correspond to the function's output dimensions while the columns correspond to the input dimensions. Robotics often involves vector valued functions, such as for dynamics. Below, we explore a simple pendulum system, frequently used to test controllers:




$$\dot{x_1} = \dot{\theta} = x_2$$
$$\dot{x_2} = \ddot{\theta} = -\frac{g}{l}\sin x_1$$

The dynamics is nonlinear, but we can linearize it by finding its Jacobian.

Similar to the question above, implement the following

```
#include <math.h>
#include <iostream>
#include <eigen3/Eigen/Eigen>

template <typename Function>
Eigen::MatrixXd calculateJacobian(Function f, Eigen::VectorXd x, double h) {
    // assumes f is a vector function with vector input
    // thus Jacobian has the dimension m x n, where f(x) in R^m and x in R^n
    // --- Your code here
```

```
    // ---
}

Eigen::VectorXd pendulumDynamics(Eigen::VectorXd x) {
    auto g = 9.81;
    auto l = 0.1;   // length
    // state space is angle and angular velocity
    Eigen::VectorXd xdot(x.rows());
    // \dot{x_0} = x_1
    xdot(0) = x(1);
    // \dot{x_1} = angular acceleration = - g / l sin(x_0)
    xdot(1) = - g / l * sin(x(0));
    return xdot;
}

int main() {
    Eigen::Vector2d x;
    x << 0., 0.;

    auto h = 0.001;
    auto J = calculateJacobian(pendulumDynamics, x, h);
    std::cout << J << std::endl;
}
```

CGAL tutorial [https://doc.cgal.org/latest/QP_solver/index.html](https://doc.cgal.org/latest/QP_solver/index.html)

Install CGAL if you don’t have it with `sudo apt-get install libcgal-dev`

CGAL has a quadratic program (QP) solver - many problems can be formulated or approximated as QPs