Series 0

Numerical methods for PDEs

2017-02-20

This is a warmup problem. You do not need to hand in this problem.

Exercise 1 Midpoint rule

In this exercise we let $a < b$ be two real numbers and $f : [a, b] \to \mathbb{R}$ be a smooth function. Our goal is to approximate the integral

$$I(f) := \int_a^b f(x)\,dx.$$

Recall that for a given number of subintervals n, the *midpoint rule* $I_n(f)$ is given as

$$I_n(f) := \frac{b-a}{n}\left[\sum_{k=0}^{n-1} f(a + (k + 1/2)\frac{b-a}{n})\right].$$

It can be checked that the error scales as $\mathcal{O}(n^{-2})$, in other words

$$|I(f) - I_n(f)| \leq Cn^{-2}.$$

a. Write a function in C++ that computes and returns (as `double`) the midpoint rule. Use the following signature

```
#pragma once
///
/// This is the type of a function taking as parameter a double, and
/// return a double
///
typedef double(*FunctionPointer)(double);

///
/// Computes the midpoint rule to approximate the integral
///
/// \param a the left endpoint
/// \param b the right endpoint
/// \param n the number of subintervals to use
/// \param f the function to compute the integral over
///
double midpoint_rule(double a, double b, int n, FunctionPointer f);
```

See `midpoint/midpoint/midpoint.cpp` for a template.

Solution:

```cpp
#include "midpoint.hpp"

double midpoint_rule(double a, double b, int n, FunctionPointer f) {
    //// NPDE_START_TEMPLATE
    double sum = 0;
    const double h = (b - a) / n;
    for(int i = 0; i < n; ++i) {
        sum += f(a + (i+0.5) * h);
    }
    return sum * h;
    //// NPDE_RETURN_TEMPLATE
    //// NPDE_END_TEMPLATE
}
```

b. **For the rest of the problem**, we set

$$a = 0.2, b = 1.3, \text{ and } f(x) = \sin(\pi x).$$

Compute the exact integral $I(f)$.

Solution:

$$I(f) = \int_a^b f(x)\,dx = \left[-\frac{\cos(\pi x)}{\pi}\right]_a^b = \frac{\cos(\pi a) - \cos(\pi b)}{\pi} \approx 0.44461596415$$

c. Write a C++ program that computes and prints $I_n(f)$ for $n = 100$. You may use the template found in `midpoint/test_single/test_single.cpp`.

Solution:

```cpp
// To use our previously written midpoint rule function
#include "midpoint.hpp"

// For printing to the terminal
#include <iostream>

// On some platforms we need to add this in order
// to get M_PI defined
#define _USE_MATH_DEFINES

// for our usual math functions and constants
#include <math.h>

// We use these two to set the precision of our output.
#include <limits>
#include <iomanip>

double f(double x) {
    return sin(M_PI * x);
}

int main(int, char**) {
    // TODO: Compute the proper approximation here:
    //// NPDE_START_TEMPLATE
    const double In = midpoint_rule(0.2, 1.3, 100, f);
    //// NPDE_END_TEMPLATE

    // Set high precision for output, easier to see what is going on
       ↪ :
    std::cout << std::setprecision(std::numeric_limits<long double
       ↪ >::digits10 + 1);

    // We print out the value of the midpoint rule here:
    std::cout << "In = " << In << std::endl;

    return 0;
}
```

and we get the output

```
./test_single
In = 0.4446380886879671146
```

which is close to the actual value of the integral.

d. In this exercise we will investiage the experimental order of convergence for the midpoint rule. Write a C++ program that computes the difference

$$|I(f) - I_n(f)|,$$

for

$$n = 2^k \qquad k = 4, 5, \ldots, 11.$$

Store the output to file and plot the results in MATLAB/Python using log scales on both aces. How does this plot agree with the error bound

$$|I(f) - I_n(f)| \leq Cn^{-2}?$$

See `midpoint/test_convergence/test_convergence.cpp` for a template.

Solution:

```
#include "midpoint.hpp" // To use our library
#include "writer.hpp" // This is the output function to write to
   ↪ file

// We store our results in a vector
#include <vector>
```

```cpp
// On some platforms we need to add this in order
// to get M_PI defined
#define _USE_MATH_DEFINES

// for our usual math functions and constants
#include <math.h>

double f(double x) {
    return sin(M_PI * x);
}


int main(int, char**) {
    //// NPDE_START_TEMPLATE
    const double a = 0.2;
    const double b = 1.3;
    double exact = (cos(M_PI*a) - cos(M_PI * b)) / M_PI;

    // We store the errors in this vector
    std::vector<double> errors;

    // We store the number of subintervals we have used here
    std::vector<int> numberOfSubintervals;

    for(int k = 4; k <= 11; k++) {
        int n = 1 << k;

        // Compute the midpoint rule here.
        double In = midpoint_rule(a, b, n, f);
        // Compute the correct error:
        double error = fabs(In - exact);
        errors.push_back(error);
        numberOfSubintervals.push_back(n);
    }

    // Write result to disk
    writeToFile("series0_1_d_errors.txt", errors);
    writeToFile("series0_1_d_numbers.txt", numberOfSubintervals);

    //// NPDE_END_TEMPLATE

    return 0;
}
```

We see from the plots that we get the desired order of convergence.

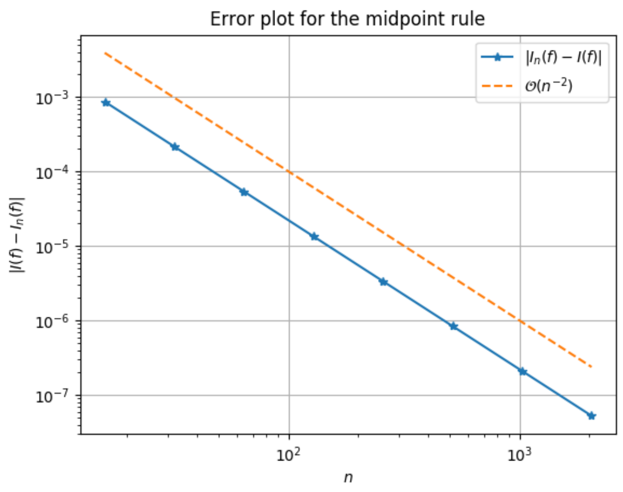

Figure 1: Convergence of the midpoint rule

Exercise 2 Linear regression

In order to detect heart diseases in cats, a biologist asks us to predict the weight of cats' hearts ($\mathbf{Y}$) with their body weight ($\mathbf{X}$). We consider the following data [1]

Body weight (kg)	2	2.2	2.4	2.2	2.6	2.2	2.4	2.4	2.5	2.7	2.6	2.2	2.5	2.5	2.5
Heart weight (g)	6.5	7.2	7.3	7.6	7.7	7.9	7.9	7.9	7.9	8.0	8.3	8.5	8.6	8.8	8.8

and propose the next linear regression

$$\mathbf{Y} = \beta_1 \mathbf{X} + \beta_0, \tag{1}$$

where $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{15}$ are the (column) vectors containing the cats' body and heart weights, repectively.

a. Use the `Eigen` Library to write a `C++` code that finds the coefficients β_0 and β_1 by solving the least square problem:

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^2} \|\mathbf{Y} - \mathbf{A}\boldsymbol{\beta}\|, \tag{2}$$

with $\boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix}$, and $\mathbf{A} = \begin{pmatrix} 1 & X_1 \\ 1 & X_2 \\ \vdots & \vdots \\ 1 & X_{15} \end{pmatrix}$.

Hint: Remember from your linear algebra lecture that this boils down to solve the associated normal equation $\mathbf{A}^T\mathbf{A}\boldsymbol{\beta} = \mathbf{A}^T\mathbf{Y}$.

Hint: The `Eigen` LU solver might be of use.

Solution:

[1] adapted from the dataset `cats` in R.

```
#include <iostream>
#include <Eigen/Dense>


int main(int argc, char **argv){

  // Declare Eigen vector type for doubles
  using vector_t =  Eigen::VectorXd ;

  // Initialize Eigen vector containing body weight in Kg(X)
  vector_t X(15);
  X << 2 , 2.2 , 2.4 , 2.2 , 2.6 , 2.2 , 2.4 , 2.4 , 2.5 , 2.7 , 2.6
     ↪  , 2.2 ,
      2.5 , 2.5 , 2.5 ;

  // Initialize Eigen vector containing heart weight in g (Y)
  vector_t Y(15);
  Y << 6.5 , 7.2 , 7.3 , 7.6 , 7.7 , 7.9 , 7.9 , 7.9 , 7.9 , 8.0 ,
     ↪ 8.3, 8.5 ,
      8.6 , 8.8 , 8.8;

  // TODO: Initialize Eigen Matrix A
  Eigen::MatrixXd A(15,2);
  //// NPDE_START_TEMPLATE
  vector_t aux(15); aux.setOnes();
  A << aux, X;
  //// NPDE_END_TEMPLATE

  // Create LHS = A'*A
  Eigen::MatrixXd LHS = A.transpose()*A;

  // TODO: Create RHS = A'*Y
  //// NPDE_START_TEMPLATE
  vector_t RHS = A.transpose()*Y;
  //// NPDE_END_TEMPLATE

  // TODO: Solve system and output coefficients b_0 and b_1
  //// NPDE_START_TEMPLATE
  Eigen::Vector2d sol = LHS.lu().solve(RHS);
  std::cout << "b_0 = " << sol(0) << " and b_1 = " << sol(1)  << std
     ↪ ::endl;
  //// NPDE_END_TEMPLATE

  return 0;
}
```

and we get the output

```
./regression
b_0 = 3.57406 and b_1 = 1.81864
```

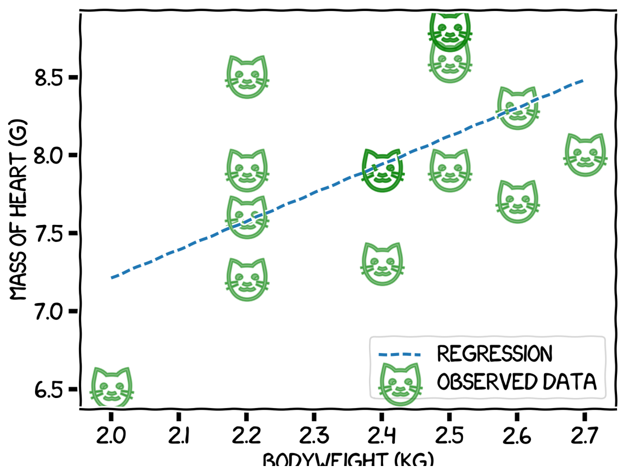

Figure 2: Comparison of regression and actual data. As we can tell, it's not a good fit.