Series 3

Advanced Numerical Methods for CSE
Last edited: December 18, 2019
Due date: January 13, 2019

Template codes are available on the course's webpage at `https://metaphor.ethz.ch/x/2019/hs/401-4671-00L/`.

Exercise 1 Riemann problem for the 1D Shallow Water Equations

In this exercise, we are going to find explicit, exact solutions to the **Riemann Problem for the Shallow Water Equations** (SWE).

Let $D \subset \mathbb{R}$, and fix a time horizon $T > 0$. The 1D SWE model the behaviour of water moving along a channel. We assume that we can represent the velocity of the fluid as a function $v : D \times [0, T] \to \mathbb{R}$; i.e. for a surface-to-bottom section of the fluid, the velocity is assumed to be constant. This is a reasonable approximation as long as the water column at any given point is not deep – hence the name.

We denote by $\tilde{h} : D \times [0, T) \to \mathbb{R}$ the height of the water column, and $\tilde{m} := \tilde{h}\tilde{v}$ the momentum of the same. If we assume a flat channel, $\forall (x, t) \in D \times (0, T)$, the equations take the form:

$$\partial_t \tilde{h}(x,t) + \partial_x \tilde{m}(x,t) = 0 \tag{1}$$

$$\partial_t \tilde{m}(x,t) + \partial_x \left(\frac{1}{2} g \tilde{h}^2(x,t) + \frac{\tilde{m}^2(x,t)}{\tilde{h}(x,t)} \right) = 0, \tag{2}$$

where g is the constant acceleration of gravity. For our convenience, we make the renormalization $h := g\tilde{h}$, $m := g\tilde{m}$, to obtain the simplified form:

$$\partial_t h(x,t) + \partial_x m(x,t) = 0 \tag{3}$$

$$\partial_t m(x,t) + \partial_x \left(\frac{1}{2} h(x,t)^2 + \frac{m(x,t)^2}{h(x,t)} \right) = 0 \tag{4}$$

We assume throughout that

$$h(x,t) > 0 \quad \forall (x,t) \in D \times [0,T),$$

since (3)-(4) are ill-defined for $h = 0$.

We will use the notation $\mathbf{U} : D \times [0,T] \to \mathbb{R}^2$,

$$\mathbf{U} := \begin{bmatrix} h \\ m \end{bmatrix}.$$

Assume that $\forall t \in [0,T)$, the solution $\mathbf{U}(\cdot,t)$ is piecewise C^1. Find the exact **entropy solution** to initial value problem for the SWE (3)-(4) with Riemann initial data:

$$\mathbf{U}(x,0) = \begin{cases} \mathbf{U}_L := \begin{bmatrix} h_L \\ m_L \end{bmatrix} & \text{if } x < 0 \\ \mathbf{U}_R := \begin{bmatrix} h_R \\ m_R \end{bmatrix} & \text{if } x > 0 \end{cases}, \tag{5}$$

for

$$h_L = 1, \qquad m_L = 1,$$

and $(h_R, m_R) =$...

a) $(4, 12)$, **b)** $(2, 2-\sqrt{3})$, **c)** $(2, 2)$.

Exercise 2 Uncertainty Quantification and Machine Learning for Finite Volume systems in 1D

We return to a generic 1D system of hyperbolic conservation laws.

Let $m \in \mathbb{N}$ and $a, b, T \in \mathbb{R}$, such that $a < b$ and $T > 0$. Then consider

$$\partial_t \mathbf{U} + \partial_x \mathbf{F}(\mathbf{U}) = 0, \qquad \forall (x,t) \in [a,b] \times (0,T), \tag{6}$$

where $\mathbf{U} : [a,b] \times [0,T] \to \mathbb{R}^m$ and $\mathbf{F} : \mathbb{R}^m \to \mathbb{R}^m$.

To complete the Cauchy problem, we need to provide the initial condition; in this exercise, we will work with **non-deterministic initial conditions**. Let $(\Omega, \Sigma, \mathbb{P})$ be a probability space. We consider initial data of the form:

$$\mathbf{U}(x, 0; \omega) := \mathbf{U}_0(x; \omega), \quad \forall (x, \omega) \in [a,b] \times \Omega. \tag{7}$$

It should be clear that, when the initial condition is non-deterministic, one can not expect the solution of eq. (6) to be deterministic. In other words, for a fixed $\omega \in \Omega$, we will obtain a solution $\mathbf{U}_\omega : [a,b] \times [0,T] \to \mathbb{R}^m$, such that $\mathbf{U}_\omega$ satisfies eq. (6) with initial data $\mathbf{U}_\omega(x, 0) = \mathbf{U}_0(x; \omega)$.

Now, let us consider an **observable** (or **functional**) of the form

$$L_T(\mathbf{U}) := \int_a^b g(\mathbf{U}(x,T)) \, dx,$$

where $g : \mathbb{R}^m \to \mathbb{R}$. Many physical quantities of interest can be cast into this form. Note that for non-deterministic initial data, the observable $q := L_T(\mathbf{U}_\omega)$ is a random variable.

Remark: In this exercise, we are going to deal with Monte Carlo (MC) and multilevel Monte Carlo (MLMC) methodology. An interesting feature of MC methods is that they are **non-intrusive**: they can be used together with any approximation algorithm (say, FVM) without modifying the underlying solver. Therefore, we encourage you to take **your implementation of exercise 1 in series 2 as a starting point** for this exercise.

For those who would rather start over with this exercise, we provide some templates based on the sample solution to exercise 2. Compared to the published solution, however, we need to make some small changes for convenience:

- `TimeLoop::operator()` now returns a `Eigen::MatrixXd` with the final state.
- `SimulationTime` now has a method `reset()` that reinitializes the object; `TimeLoop::operator()` begins by calling it.
- Fixed γ to the typical value of 1.4 in `Model::Euler`.
- Added the domain limits, xL and xR, to the json config file.
- Combined all models into a single `fvm_uq.cpp` file.
- We focus on FVM for this exercise; the extension to DG is straightforward.

(ML)MC recap Fix some observables $\{L_T^j\}_{j=1}^J$, $J \in \mathbb{N}$. We would like to find statistics of the random variable $q := \left(L_T^j(\mathbf{U}_\omega)\right) \in \mathbb{R}^J$. For simplicity of the exposition we fix $J = 1$ below.

Let $k \in \mathbb{N} \cup \{0\}$ be the number of random parameters that the initial condition requires: $k = 0$ in the case of deterministic initial data, and $k > 0$ otherwise[1].

When $k > 0$, let $Y : \Omega \to \mathbb{R}^k$ be a random variable, $Y \sim (\mathcal{U}[0,1])^k$. That is, Y has k components, independent and identically distributed, each following a uniform distribution of values in [0,1]. By a slight abuse of notation, we henceforth identify Y and ω, and denote $\boldsymbol{\omega} \sim (\mathcal{U}[0,1])^k$.

We parameterize[2] our initial condition as $\mathbf{U}_0(\cdot;\boldsymbol{\omega})$.

MC We generate samples $\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \ldots, \boldsymbol{\omega}_M$, and denote $q_i := L_T(\mathbf{U}^h_{\boldsymbol{\omega}_i})$, where $\mathbf{U}^h_{\boldsymbol{\omega}_i}$ is an approximation of $\mathbf{U}_{\boldsymbol{\omega}_i}$, computed with a mesh indexed by grid size h. We approximate the probability distribution of $q := L_T(\mathbf{U}_{\boldsymbol{\omega}})$ as

$$\mathrm{Law}(q) \approx \frac{1}{M}\sum_{i=1}^{M} \delta_{q_i},$$

where δ_v is the Dirac delta distribution, which takes value v with probability 1. For any function f, e.g. $f \equiv id$, we have $\mathbb{E}(f(q)) \approx \frac{1}{M}\sum_{i=1}^{M} f(q_i)$.

MLMC Fix a maximum number of levels n_l. For $l \in \{0, 1, \ldots, n_l\}$, let q_l be random variables which approximate q in some sense. Then, using the telescopic sum, we can write

$$q_{n_l} = q_0 + (q_1 - q_0) + (q_2 - q_1) + \cdots + (q_{n_l} - q_{n_l - 1}).$$

We refer to variables $q_l - q_{l-1}$ as **increments** or **details**.

For each level $l \in \{0, 1, \ldots, n_l\}$, fix a number of samples M_l, and a grid size h_l. We generate a set of samples $\{\boldsymbol{\omega}_i^l\}_{l=0, i=1}^{l=n_l, i=M_l}$, and denote

$$q_i^{l,m} := L_T(\mathbf{U}^{h_l}_{\boldsymbol{\omega}_i^m}).$$

Here the first super-index indexes the grid, and the second indexes the level for the sample; either $m = l$ or $m = l + 1$. We approximate the probability distribution of q as:

$$\mathrm{Law}(q) \approx \frac{1}{M_0}\sum_{i=1}^{M_0} \delta_{q_i^{0,0}} + \sum_{l=1}^{n_l}\frac{1}{M_l}\sum_{i=1}^{M_l}\left(\delta_{q_i^{l,l}} - \delta_{q_i^{l-1,l}}\right).$$

For any function f, we have

$$\mathbb{E}(f(q)) \approx \frac{1}{M_0}\sum_{i=1}^{M_0} f(q_i^{0,0}) + \sum_{l=1}^{n_l}\frac{1}{M_l}\sum_{i=1}^{M_l}\left(f(q_i^{l,l}) - f(q_i^{l-1,l})\right).$$

[1]See task 1a) for examples

[2]If distributions other than $\mathcal{U}[0,1]$ are required, the approach of inverse transform sampling can be applied

Important: Note that $q_i^{l,l} - q_i^{l-1,l} = L_T(\mathbf{U}^{h_l}_{\boldsymbol{\omega}_i^l}) - L_T(\mathbf{U}^{h_{l-1}}_{\boldsymbol{\omega}_i^l})$; i.e., both realizations use the same sample $\boldsymbol{\omega}_i^l$, but different grid sizes h_l and h_{l-1}.

2a)

Templates: include/ancse/initial_condition.hpp, src/ancse/initial_condition.cpp.

We have defined a class `InitCond` that implements the initial conditions. This includes

- `int get_num_rand_params()`, which returns the number k of random parameters needed (0 for deterministic initial data).
- `Eigen::MatrixXd operator()(const Grid &grid, std::vector<double> omega)`, which evaluates the initial condition given by a vector `omega` of random parameters $\boldsymbol{\omega} \sim (\mathcal{U}[0,1])^k$ on grid `grid`.

You can follow as an example the `class SodShockTubeShifted`, which implements the following initial condition, with $k = 1$ random parameter:

$$\mathbf{U}_0(x) := \begin{cases} \mathbf{U}_L & \text{if } x < \epsilon G(\omega) \\ \mathbf{U}_R & \text{if } x > \epsilon G(\omega) \end{cases}, \quad \mathbf{U}_L := \begin{pmatrix} \rho_L \\ v_L \\ p_L \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \quad \mathbf{U}_R := \begin{pmatrix} \rho_R \\ v_R \\ p_R \end{pmatrix} = \begin{pmatrix} 0.125 \\ 0 \\ 0.1 \end{pmatrix}, \tag{8}$$

with $\epsilon = 0.1$ and $G(y) = 2y - 1$. That is, the initial shock location follows a $\mathcal{U}[-0.1, 0.1]$ distribution.

Implement the following non-deterministic version[3] of the Sod shock tube, with 6 random parameters:

$$\mathbf{U}_0(x) := \begin{cases} \mathbf{U}_L & \text{if } x < \epsilon G(\omega_1) \\ \mathbf{U}_R & \text{if } x > \epsilon G(\omega_1) \end{cases}, \tag{9}$$

$$\mathbf{U}_L := \begin{pmatrix} \rho_L \\ v_L \\ p_L \end{pmatrix} = \begin{pmatrix} 1 + \epsilon G(\omega_2) \\ \epsilon^2 G(\omega_4) \\ 1 + \epsilon G(\omega_5) \end{pmatrix}, \quad \mathbf{U}_R := \begin{pmatrix} \rho_R \\ v_R \\ p_R \end{pmatrix} = \begin{pmatrix} 0.125(1 + \epsilon G(\omega_3)) \\ \epsilon^2 G(\omega_4) \\ 0.1(1 + \epsilon G(\omega_6)) \end{pmatrix}, \tag{10}$$

where $\epsilon = 0.1$, $G(y) = 2y - 1$, and the random parameters $\omega_i \sim \mathcal{U}[0,1]$ are independent, for $i = 1, 2, \ldots, 6$. Note the sub-indices carefully, especially for v_L and v_R.

Register your implementation in `initial_condition.cpp`.

2b)

Templates: include/ancse/functional.hpp, src/ancse/functional.cpp.

[3] K. Lye, S. Mishra, D. Ray, *Deep learning observables in computational fluid dynamics*

Implement the computation of some observables in order to study their statistics; for example, the mean density $\bar{\rho}$ and the total energy E_{total}, where

$$\bar{\rho} := \frac{1}{b-a}\int_a^b \rho(x,T)\ dx, \quad E_{total} := \int_a^b E(x,T)\ dx.$$

Additionally, implement the functionals

$$L_T^{l,r}(\mathbf{U}) := \frac{1}{r-l}\int_l^r \rho(x,T)\ dx, \qquad \text{for } (l,r) \in \{(-1.5,-0.5),(0.8,1.8),(2,3)\}$$

Register your implementation in `functional.cpp`.

2c)

Templates: include/ancse/statistics.hpp, src/ancse/statistics.cpp.

Implement a class `Statistics` to keep track of the desired statistics of any observable. Its internal data-structures and output are up to you, but it should implement the following functions:

- `add_coarse_statistics(Eigen::MatrixXd coarse)`: receives a matrix in $\mathbb{R}^{J\times M}$ (i.e. `n_functionals` rows and `n_samples` columns) of realizations $q_{j,i} := L^j(u(\cdot,T;\omega_i))$ coming from a MC simulation, and stores the desired information.
- `add_detail_statistics(Eigen::MatrixXd fine, Eigen::MatrixXd coarse, int lvl)`: receives the matrices of realizations as above of $(q_i^{l,l})_j$ (`fine`) and $(q_i^{l-1,l})_j$ (`coarse`), corresponding to the level `lvl` of a MLMC simulation, and stores the desired information.
- `output_statistics()`: uses the stored values to output some statistics of the approximation of the random variable q. These should include at least:
  - mean $\mathbb{E}(q)$,
  - $\mathrm{Var}(q)$,
  - $\mathrm{Var}(q_l - q_{l-1})$, for all $l > 1$ (MLMC only).

Hint: Recall that $\mathrm{Var}(q) = \mathbb{E}(q^2) - \mathbb{E}(q)^2$, so keeping track of statistics for q^2 would be useful.
Hint: Functionals will be read from field `functionals` in `config.json`, which should contain a comma-separated **list of names**, instead of a single name.

2d)

Template: src/fvm_uq.cpp.

Note the function generate_omegas(nparams), which generates a vector with a realization of k independent $\mathcal{U}[0,1]$ variables, and generate_all_omegas(nsamples, nparams), which generates a vector of nsamples of such vectors. These are already given to you, and are initialized so that the results of the program are reproducible by default.

Complete the function do_mlmc to implement single-level **Monte Carlo**, which will read from the JSON file the number of samples (nsamples_finest) as well as the number of cells (n_interior_cells_coarsest).

Make use of the function generate_all_omegas as well as your own make_fvm.

Compute statistics of the functionals with your class Statistics.

2e)

Template: src/fvm_uq.cpp.

Expand your implementation of do_mlmc to **multi-level Monte Carlo**, for n_levels>1.

For level l, use a mesh with 2^l * n_interior_cells_coarsest cells, and $2^{(n_l-l)}$ * nsamples_finest samples.

Compute statistics of the functional with your class Statistics. Does it hold that the variance of the details tends to zero?

2f)

An alternative to the (ML)MC framework lies in **Machine Learning** techniques.

One must start by generating a training set for the neural network. For this, complete function generate_dataset, which will run if parameter generate_dataset in config.json is set to true.

This function must generate two lists of nsamples_finest elements. Each element in the first list is a realization $\boldsymbol{\omega}_i$ of a vector of random parameters $\boldsymbol{\omega} \sim (\mathcal{U}[0,1])^k$. Each element in the second list is the corresponding evaluations of the functional $\{L_T^j(U_{\boldsymbol{\omega}_i}^h)\}_{j=1}^J$.

Print this output to a file (e.g. a JSON file); this will be your input for the next exercise.

Hint: Your Monte Carlo routine can be of use in this.

Exercise 3 Learning a Functional by Deep Neural Networks

Let $\mathbf{U} : D \times [0,T] \times \Omega \to \mathbb{R}^m$ be the solution of

$$\mathbf{U}_t + \mathbf{F}(\mathbf{U})_x = 0 \tag{11}$$

$$\mathbf{U}(x,0;\omega) = \mathbf{U}_0(x;\omega) \tag{12}$$

with $D = [-5,\, 5]$, $m = 3$, $\mathbf{U} = (\rho,\, \rho v,\, E)$ and $\mathbf{F}(\mathbf{U}) = (\rho v,\, \rho v^2 + p,\, v(E+p))$. Furthermore, let $\Omega = [0,\, 1]^k$, $k = 6$ and $\mathbf{U}_0(x;\omega)$ be the initial conditions of a Sod shock tube with random location and strength. More precisely,

$$\mathbf{U}_0(x;\omega) = \begin{cases} \mathbf{U}_L(\omega) & x < \xi(\omega) \\ \mathbf{U}_R(\omega) & \text{otherwise,} \end{cases} \tag{13}$$

where $\mathbf{U}_L$ and $\mathbf{U}_R$ are the states to the left and right of the shock defined by the values of their primitive variables:

$$\rho_L = 1 + \epsilon\, G(\omega_2), \qquad v_L = \epsilon^2 G(\omega_4), \qquad p_L = 1 + \epsilon\, G(\omega_5) \tag{14}$$

$$\rho_R = 0.125(1 + \epsilon\, G(\omega_3)), \qquad v_R = \epsilon^2 G(\omega_4), \qquad p_R = 0.1(1 + \epsilon\, G(\omega_6)), \tag{15}$$

with $\epsilon = 0.1$, $G(\omega) = 2\omega - 1$. Note that $v_L = v_R$ and $\xi(\omega) := \epsilon\, G(\omega_1)$.

For $j = 1, \ldots 3$, let

$$\mathcal{L}_j : \Omega \to \mathbb{R}, \quad \omega \mapsto \int_{I_j} \rho(x,T;\omega)dx, \tag{16}$$

where $I_1 = [-1.5,\, -0.5]$, $I_2 = [0.8,\, 1.8]$ and $I_3 = [2,\, 3]$, and ρ be the density of the solution to the Euler equations. Furthermore, let $\mathcal{L} := (\mathcal{L}_1,\, \mathcal{L}_2,\, \mathcal{L}_3)$.

The aim of this exercise is to study the approximation of $\mathcal{L}$ by a Deep Neural Network (DNN).

3a)

Define a suitable architecture of your neural network, e.g. number of layers, number of neurons per layer and the activation function of each layer.

Also explicitly write down the neural network as a concatenation of non-linear and affine transformations.

3b)

Formulate both the continuous optimization problem which is referred to as "training the neural network", and the discrete version of the optimization problem which is solved in practice.

3c)

Train a neural network which approximates $\mathcal{L}$. You are trying to achieve a relative L^2-error of a few percent.

3d)

Use this example to familiarize yourself with the concepts related to DNNs.

For example, by answering the following questions:

- What is Adam's optimizer?
- What is "batch size" and how does it affect the discrete version of the optimization problem?
- What are "epochs", and what is their connection to the number of steps in the optimization?
- How does the batch size affect the runtime of the training? How should one choose the batch size?
- What are the features this problem has which make us interested in solving it using DNNs?
- How does the accuracy achieved by approximating $\mathcal{L}$ by DNNs compare with traditional methods?

Some of these questions are rather high-level and do not have precise mathematical answers (yet).