Input handling and functions

Paolo Joseph Baioni
paolojoseph.baioni@polimi.it

March 7, 2025

Warning

To complete some points of the following exercises you need to install the utilities in "Examples Utilities" and the "Extras". If the submodules folders (json and muparser) are empty, you forgot the `--recursive` option when cloning the pacs repo, you can fix it by running `git submodule update --init` in the root folder of the pacs repo.
If you are using the docker (or podman) container, this has been already set-up for you.
Otherwise, if you haven't, you can follow the next slide steps (from the pacs-examples repo READMEs).

Steps

Assuming you have already installed mk modules, eg

```
wget https://github.com/pcafrica/mk/releases/download/v2024.0/mk-2024.0-full.tar.
    gz
sudo tar xvzf mk-2024.0-full.tar.gz -C /
```

or you are using them through apptainer, to set up a clean version of the repos from scratch you can:

```
git clone git@github.com:pacs-course/pacs-Labs
git clone --recursive git@github.com:HPC-Courses/pacs-examples.git --branch master
cd pacs-examples
source load_modules.sh
cd Extras
./install_extras.sh
cd ../Examples
cp Makefile.user Makefile.inc
pwd
#modify PACS_ROOT to be pwd in Makefile.inc
bash ./setup.sh
```

Exercise 1 - Newton solver

1. Implement a `NewtonSolver` class.
2. Let algorithm parameters be read from the command line using `GetPot`.
3. Write different `main` files that pass functions and derivatives as:
   - 3.1 function pointers;
   - 3.2 (*Homework*) lambda functions;
   - 3.3 `muParser` functions (the `muParser` library can be installed by running `./install_PACS.sh` from `${PACS_ROOT}/Extras/muParser`).
   - 3.4 (*Homework*) Let the user pass the function and the parameters from command line using `GetPot` and `muParser`.
4. Use the solver to solve the equation

$$x^3 + 5x + 3 = 0.$$

Exercise 2 - Horner algorithm

Evaluating high order polynomial can be computationally expensive in terms of floating point operations.

Given a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$, instead of directly evaluating all the monomials at x_0, we compute the following sequence:

$$\begin{aligned} b_n &= a_n, \\ b_{n-1} &= a_{n-1} + b_n x_0, \\ &\vdots \\ b_0 &= a_0 + b_1 x_0. \end{aligned}$$

The Horner rule exploits the already computed informations to reduce the number of floating point operations.

Curiosity: *Horner's rule is optimal when evaluating scalar polynomials sequentially.*

Exercise 2 - Horner algorithm

1. Implement the `eval()` and `eval_horner()` functions to compute:

$$p_{\mathrm{eval}}(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_n x^n,$$
$$p_{\mathrm{Horner}}(x) = a_0 + x\left(a_1 + x\left(a_2 + x\left(a_3 + \ldots + x\left(a_{n-1} + x\, a_n\right) \ldots\right)\right)\right).$$

2. Implement an `evaluate_poly()` function by manually looping over the input points.
3. Modify `evaluate_poly()` to use `std::transform`.
4. Implement an `evaluate_poly_parallel()` that makes use of the parallel execution policies of `std::transform` (available since `C++17`).
5. Convert `eval` and `eval_horner` from function pointers to `std::function`.
6. (*Homework*) Let the user choose from a json parameters file the degree of the polynomial and the discretization interval. The `json` library can be installed by running `./install_PACS.sh` from `${PACS_ROOT}/Extras/json`

NB: the parallel version requires to link against the Intel Threading Building Blocks (`TBB`) library (preprocessor flags `-I${mkTbbInc}`, linker flags `-L${mkTbbLib} -ltbb`).