# Input handling and functions

Alberto Artoni

13/03/2024

## 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.

## GetPot

GetPot is a header only library. It allows to easily pass arguments from command line and to read parameter files.

Let us see some example...

## Exercise 1 - Newton solver

Find the root of the following non linear equation:

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

by employing the `NewtonSolver.hpp` library.

1. Customize the solver by passing the parameters from command line using `GetPot`.
2. Write a new `main` file that pass functions and derivatives as `muParser` functions.
3. (*Homework*) Write a new `main` file that pass the function and the parameters from command line using `GetPot` and `muParser`.

**Note:** Find the `muParser` library in `${PACS_ROOT}/lib` and the header file in `${PACS_ROOT}/include`.

## 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()` function that computes $p(x)$ by evaluating explicitly the monomials:

$$p_{\text{eval}}(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_n x^n.$$

2. Implement `eval_horner()` function to compute $p(x)$ with the Horner's rule:

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

3. Implement an `evaluate_poly()` function by manually looping over the input points.
4. Modify `evaluate_poly()` using the function `std::transform`.
5. Implement an `evaluate_poly_parallel()` that makes use of the parallel execution policies of `std::transform` (available since `C++17`).
6. (*Homework*) Let the user choose from a `json` parameters file the degree of the polynomial and the discretization interval.

**Note**:

- the parallel version requires to link against the Intel Threading Building Blocks (`TBB`) library (preprocessor flags `-I${mkTbbInc}`, linker flags `-L${mkTbbLib} -ltbb`).
- Find the `json` header file in `${PACS_ROOT}/Examples/include`.