Uncertainty Quantification and Quasi-Monte Carlo Exercise 6
Wintersemester 2022/23
Return your written solutions either in person or by email
to vesa.kaarnioja@fu-berlin.de by Tuesday 29 November, 2022, 12:15

Please note that there are a total of 3 tasks this week!

1. Let $K\colon [0,1]\times[0,1]\to\mathbb{R}$ be defined by $K(x,y)=\min\{x,y\}$. Show that

$$\lambda_k = \frac{1}{(k-\frac{1}{2})^2\pi^2} \quad \text{and} \quad \psi_k(x) = \sqrt{2}\sin((k-\tfrac{1}{2})\pi x), \quad k=1,2,3,\ldots,$$

are the eigenvalues and eigenfunctions of

$$\int_0^1 K(x,y)\psi_k(y)\,\mathrm{d}y = \lambda_k\psi_k(x) \tag{1}$$

for all $k=1,2,3,\ldots$.
Hint: Differentiating the integral equation (1) twice on both sides with respect to x should yield an ODE of the form $\lambda\psi''(x)+\psi(x)=0$. This is a second order ODE with constant coefficients, which has a general solution of the form $\psi(x)=A\sin\left(\frac{x}{\sqrt{\lambda}}\right)+B\cos\left(\frac{x}{\sqrt{\lambda}}\right)$, $A,B\in\mathbb{R}$.

2. Let us consider the high-dimensional integral

$$I_s := \int_0^1\cdots\int_0^1 \cos\left(2\pi+\sum_{i=1}^s x_i\right)\mathrm{d}x_1\cdots\mathrm{d}x_s.$$

Estimate the value of this integral by implementing a Monte Carlo sampler in your favorite programming language. That is, compute the sample average

$$Q_{s,n}(f) = \frac{1}{n}\sum_{k=1}^n f(\boldsymbol{t}_k), \quad f(\boldsymbol{x}) := f(x_1,\ldots,x_s) := \cos\left(2\pi+\sum_{i=1}^s x_i\right),$$

where $\boldsymbol{t}_1,\ldots,\boldsymbol{t}_n$ are drawn from the uniform distribution $\mathcal{U}([0,1]^s)$.

In this case, the exact value of this integral is $I_s = 2^s\cos\left(2\pi+\frac{s}{2}\right)\sin(\frac{1}{2})^s$ (you do not need to prove this). Compute the error $|I_s - Q_{s,n}(f)|$ for $n=2^k$, $k=0,1,2,\ldots,20$. Try out several values for the dimension s, for example, $s=10,100,1\,000,\ldots$. What convergence rate do you observe for the error as a function of n? Does increasing the dimension s affect the convergence rate?
Please make sure to return the source code you used to solve this task.
MATLAB users: `rand(m,n)` produces an $m\times n$ array containing uniformly distributed random numbers between 0 and 1. The commands `sum(a,1)` and `sum(a,2)` can be used to compute the column and row sums of array `a`, respectively.
Python users: the numpy package contains the following helpful functions:
`numpy.random.uniform(low=0.0, high=1.0, size=(m,n))`
`numpy.sum(a,axis=int)`
where the options `int` $=$ `0` and `int` $=$ `1` correspond to computing the column and row sums of array `a`, respectively.

3. Let us try out *parallel computing* through a *very* simple benchmark problem. Using either MATLAB (if you have the Parallel Computing Toolbox installed) or Python, create a function (let's call the function `foo`) which does the following:

   (i) Each time the function is called, a new 150×150 matrix `A` and a new 150×1 vector `y` are created, both containing uniformly distributed random numbers between 0 and 1 as entries.

   (ii) The vector `x` is solved from the linear system $\texttt{A}\texttt{x} = \texttt{y}$.

   (iii) As output, the function returns the sum $\texttt{x}_1 + \texttt{x}_2 + \cdots + \texttt{x}_{150}$.

   The function does not need to take any arguments as input (**except, maybe, if you follow the Python instructions below**). This is a very simple example of an "embarrassingly parallel problem".

   First, using an ordinary `for` loop, call the function, e.g., 10 000 times. Record the time it takes to complete this task.

   Next, instead of using a sequential `for` loop, try repeating this same task by executing the same program 10 000 times in *parallel*. Record the time it takes to complete this task. Did you gain any improvement over the sequential `for` loop? *Please make sure to return the source code you used to solve this task.*

   Below are some tips on how to make this work.

   MATLAB users (if the Parallel Computing Toolbox is installed): You can create a new *parallel pool* by executing the commands

```
poolobj = gcp('nocreate');
if isempty(poolobj)
   parpool;
end
```

   Instead of a regular `for` loop, you can now execute a `for` loop in *parallel* by running

```
parfor k = 1:10000
   foo;
end
```

   You can record the time it takes to execute a code segment using the commands

```
tic; % start the timer
<insert the script you wish to time here>
time = toc % stop the timer and record the time (in seconds)
```

   Other helpful commands include `rand(m,n)` and `sum` (see task 2 above) as well as `mldivide` or the backslash operator \ which can be used to solve a linear system.

   Python users: There are several good options to do parallel programming in Python, but here is a fairly simple method. First, make sure that the `joblib` package is installed.[1] In the preamble, you will need something like

[1] Installation can be done via `pip install joblib` or `pip3 install joblib` as before.

```
import numpy
import time
from joblib import Parallel, delayed
```

To implement the parallel `for` loop, you can execute the following

```
n_cpus = <insert number of CPU cores here, e.g., 2>
results = Parallel(n_jobs=n_cpus)(delayed(foo)(i) for i in range(10000))
```

Note that the function `foo` needs to take as input a "dummy" variable `i` for the above command to work (the variable `i` does not need to do anything in the actual function).

You can record the time it takes to execute a code segment using the commands

```
t0 = time.time()
<insert the script you wish to time here>
t1 = time.time() # stop the timer
time_total = t1-t0; # record the time (in seconds)
```

Other helpful commands include `numpy.random.uniform` and `numpy.sum` (see task 2 above) as well as `numpy.linalg.solve`, which can be used to solve a linear system.

Remark: Depending on your platform / implementational details in Python, it may be necessary to wrap your program inside
`if __name__ == '__main__':`

If you are unable to get the parallelization to work, don't worry too much! Feel free to only implement the sequential algorithm in this case.