Scientific Programming 22 (2014) 37–51
DOI 10.3233/SPR-140377
IOS Press

Python for scientific computing education: Modeling of queueing systems

Vladimiras Dolgopolovas [a,*], Valentina Dagienė [a], Saulius Minkevičius [b] and Leonidas Sakalauskas [b]

[a] *Informatics Methodology Department, Institute of Mathematics and Informatics of Vilnius University, Vilnius, Lithuania*

[b] *Operational Research Sector at System Analysis Department, Institute of Mathematics and Informatics of Vilnius University, Vilnius, Lithuania*

Abstract. In this paper, we present the methodology for the introduction to scientific computing based on model-centered learning. We propose multiphase queueing systems as a basis for learning objects. We use Python and parallel programming for implementing the models and present the computer code and results of stochastic simulations.

Keywords: Scientific computing education, model-centered learning, multiphase queueing systems, parallel programming, Python

1. Background and introduction

In this study, we understand the meaning of "scientific computing" as using computers to analyze and solve scientific and engineering problems. We distinguish that from pure numerical computations. Studying scientific computing is always a challenging task for the learner as well as for the educator. Such a studying process deals with plenty of technical and multidisciplinary issues and requires a synchronization of the learner's mathematical and computer science competencies. To overcome these difficulties, we propose a set of learning objects and a methodology, which is based on a constructivist approach to learning and provides a relevant framework for the educator. Such a framework enables the learner to conduct a series of computational experiments with computer models. Using such an approach, related mathematical and programming knowledge is provided on demand and parallel to the main curriculum. We consider a computational statistics section of the introductory scientific computing course as possible application scope of this research. Below, we provide the background of the presented methodology.

*Corresponding author: Vladimiras Dolgopolovas, Informatics Methodology Department, Institute of Mathematics and Informatics of Vilnius University, Akademijos 4, LT-08663 Vilnius, Lithuania. E-mail: vdolgopolovas@gmail.com.

1.1. Scientific computing

Karniadakis and Kirby II define "scientific computing is the heart of simulation science". The authors offer a "*seamless* approach to numerical algorithms, modern programming techniques, and parallel computing. ... Often times such concepts and tools are taught *serially* across different courses and different textbooks, and hence the interconnection between them is not immediately apparent. The necessity of integrating concepts and tools usually comes after such courses are concluded, e.g. during a first job or a thesis project, thus forcing the student to synthesize what is perceived to be three *independent subfields* into one in order to produce a solution. Although this process is undoubtedly valuable, it is time consuming and in many cases it may not lead to an effective combination of concepts and tools. Moreover, from the pedagogical point of view, the integrated seamless approach can stimulate the student simultaneously through the eyes of multiple disciplines, thus leading to enhanced understanding of subjects in scientific computing" [16]. Figure 1 presents the definition of scientific computing as an intersection of numerical mathematics, computer science and modeling [16].

1.2. Constructivist learning

Caine and Caine in their fundamental research [6] propose the main principles of constructivist learning. One of the most important for us is as follows: "The

1058-9244/14/$27.50 © 2014 – IOS Press and the authors. All rights reserved

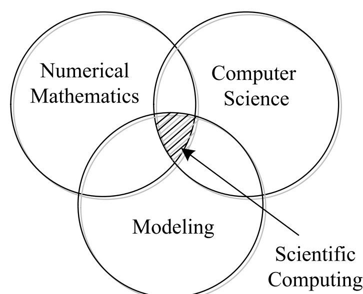


Fig. 1. Scientific computing.

brain processes parts and wholes simultaneously". So, a well-organized learning process provides details as well as underlying ideas. Using model-centered learning, we introduce the goal of the research after constructing a model for simulation. That allows us to observe the results and to draw relevant conclusions.

1.3. Model-centered education: Why models?

Gibbons introduced model-centered instruction in 2001 [9]. The following main principles are important:

- Learner's *experience* is obtained by interacting with models;
- Learner solves scientific and engineering *problems* using simulation on models;
- Problems are presented in a constructed *sequence*;
- Specific instructional *goals* are specified;
- All necessary *information* within a solution environment is provided.

Millard et al. [30] propose model facilitated learning using "interactive simulations". The authors present a modern computer technology powered by "promising methodology" based on "system dynamics". "Supportable experiences include the construction of interactive ... models as well as their use *for hypothesis testing and experimentation*".

Lehrer and Schauble [25] refer to the experiments with different representations of the model: "Student learning is enhanced when students have multiple opportunities to invent and revise models and then to compare the explanatory adequacy of different models".

1.4. Scientific computing education: Experiments with models

Xue [40] introduces "teaching reform ideas in the "scientific computing" education by means of modeling and simulation". He suggests "... the use of the modeling and simulation to deal with the actual problem of programming, simulating, data analyzing ...". Model-centered learning is used in mathematics education. Plenty of models are constructed using "Geogebra" software [33]. Models play a central role in Science Education [7,18].

1.5. Stochastic simulations of queueing systems

We propose queueing systems due to the simplicity of primary definitions and due to wide possibilities for modeling and simulation. The Queueing Systems Theory is well established and simulations of queueing systems are widely used in science [4,19] and education [13,36]. The multiphase queueing system is a good platform for learner experiments using parallel calculations. Also there is a number of interesting theoretical results to study and to investigate [12].

1.6. Python in scientific computing education

Python is one of the most popular programming languages for scientists as well as for educators [21–23]. Python is widely used in industrial scientific computing [14]. Langtangen reports on the long term experience of using Python as the primary language for teaching Scientific Computing in the University of Oslo [24]. Python is promoted as the first language to study programming [38] as well as for advanced studies of computational methods [3,20,34].

2. The basics

Below we present a brief on the key topics of an introductory curriculum in Scientific Computing. These topics include randomness with random numbers and distributions, stochastic simulations and multiprocessing. We use a simple model of throwing a die or a number of dice. The main task of these experiments is to provide an experimental proof of the Central Limit Theorem. These models and experiments with such models also enhance the learner's understanding of pseudo- and quasi-random number generators and the exponential distribution. That could provide basic ideas for more advanced experiments with the model of queueing systems.

2.1. Random numbers and distributions

All probability topics could be traditionally considered difficult to understand and are always within the scope of interests of international education schol-

ars [15]. At the same time such topics are very important in scientific research [10]. The model-centered approach makes it easier to understand the material. The model we are studying is a simple model of throwing a die or a specific number of dice. We start from one die and continue experimenting with more dice.

The aim of these introductory experiments is rather complex. We not only introduce probability and distributions, but also we simultaneously introduce stochastic simulations and parallel computing. We also take one step towards a scientific research as we introduce the experimental proof of the Central Limit Theorem.

We begin with the introduction of random number generators leaving distributions behind the scene. We then explain uniform random numbers. Discussions about true randomness or quasi randomness [26,35] could follow. For advanced learners, the task to carry out a number of experiments with pseudo-randomness and the Python pseudo-random module could be presented. As an introductory step, the assignment for the learner is to increase the number of trials and supervise the results of simulations. In the next steps, we proceed to more sophisticated experiments and parallel calculations. We use the Python random module for simulations and the mpi4py for parallel programming. The Python random module implements pseudo-random number generators for various distributions. For example *random.randint*(a, b) returns a random integer N such that $a \leqslant N \leqslant b$ and *random.expovariate*(lambd) returns exponentially distributed random numbers with the parameter 'lambd'. One should refer to Python documentation for specific details. The programming model for a single die is presented in Fig. 2.

The results of a simulation in the case of a single die are presented in Fig. 3.

Next, we proceed to the case of two dice. The main idea at this point is to explain the Central Limit Theorem by experimenting with different numbers of dice. Figure 4 represents this idea.

The learner proceeds by modifying the two-dice code that enables him to start a multi dice case. The code is analogical to the one die code except for the two instructions presented below:

```
...
list_of_values.append(random.randint(1, 6)
                      + random.randint(1, 6))
...
pylab.hist(list_of_values, pylab.arange(1.5, 13.5, 1.0))
...
```

```
import pylab
import random

number_of_trials = 100

## Here we simulate the repeated throwing of a single six-sided die
list_of_values = []
for i in range(number_of_trials):
    list_of_values.append(random.randint(1,6))

print "Trials =", number_of_trials, "times."
print "Mean =", pylab.mean(list_of_values)
print "Standard deviation =", pylab.std(list_of_values)

pylab.hist(list_of_values, bins=[0.5,1.5,2.5,3.5,4.5,5.5,6.5] )
pylab.xlabel('Value')
pylab.ylabel('Number of times')
pylab.show()
```

Fig. 2. Python single die model.

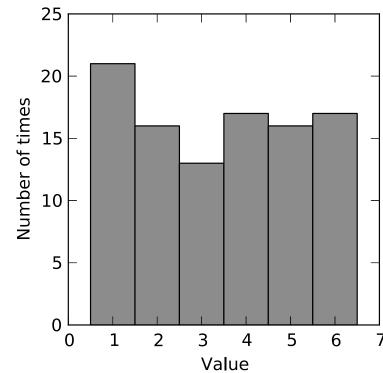


Fig. 3. Simulation results for a single die.

The results of a simulation in the two dice case are presented in Fig. 5.

We can now go on to the normal distribution. The task here is to show how the above multi-dice case correlates with the normal distribution. Another task would be to introduce the mean and deviation. The code is similar to the one-die case except for the instruction used below:

```
...
list_of_values.append(random.normalvariate(7, 2.4))
...
```

The results of a simulation for the normal distribution are presented in Fig. 6.

The final step is to introduce the exponential distribution. One always uses the exponential distribution for simulating interarrival times of customers in queue-

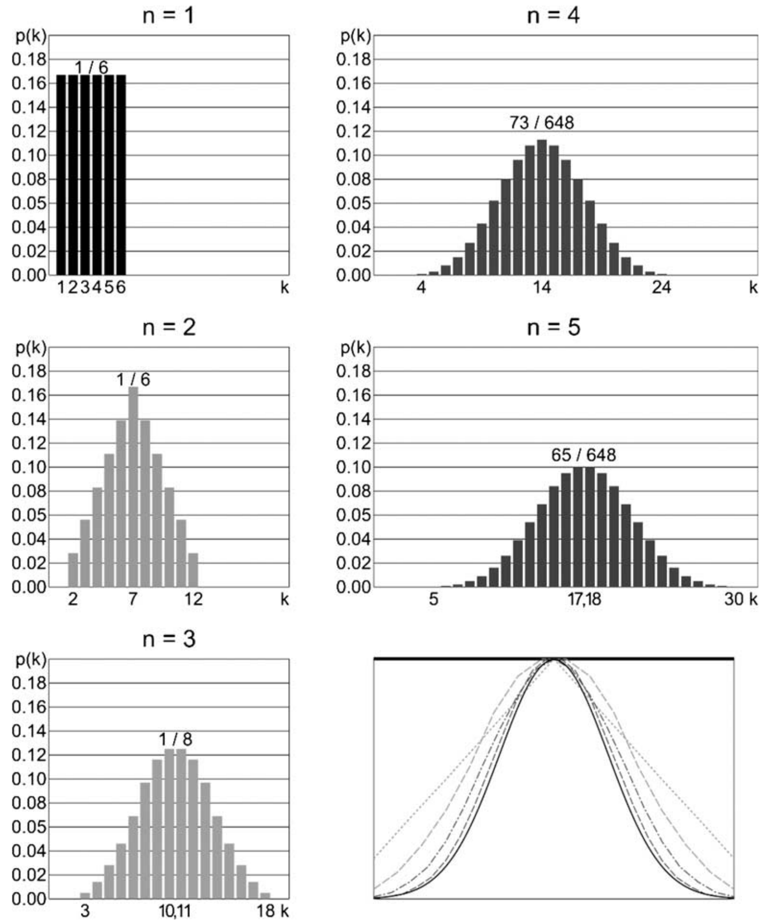


Fig. 4. Comparison of the probability density functions (from http://en.wikipedia.org/wiki/Central_limit_theorem).

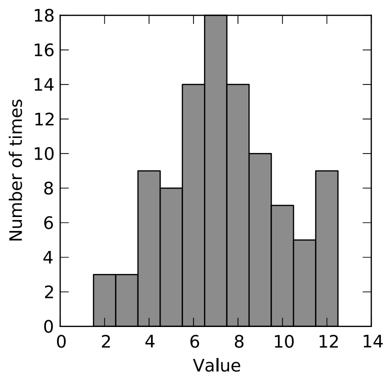


Fig. 5. Two-dice case.

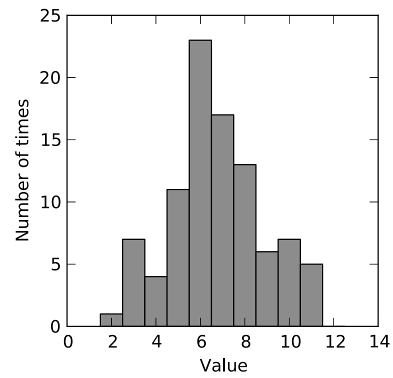


Fig. 6. Simulation results for the normal distribution.

ing systems of various types. The model of the exponential distribution and the results of a simulation are presented in Figs 7 and 8.

2.2. Stochastic simulation

Stochastic simulation is of primary importance in the field of scientific computing. We focus on Monte Carlo methods [10,11,27]. After the model has been

```
import pylab
import random

number_of_trials = 1000
number_of_customer_per_hour=10

## Here we simulate the  interarrival time of the customers

list_of_values = []
for i in range(number_of_trials):
        list_of_values.append(random.expovariate(
number_of_customer_per_hour))

mean=pylab.mean(list_of_values)
std=pylab.std(list_of_values)
print "Trials =", number_of_trials, "times"
print "Mean =", mean
print "Standard deviation =", std

pylab.hist(list_of_values,20)
pylab.xlabel('Value')
pylab.ylabel('Number of times')
pylab.show()
```

Fig. 7. Python model for the exponential distribution.

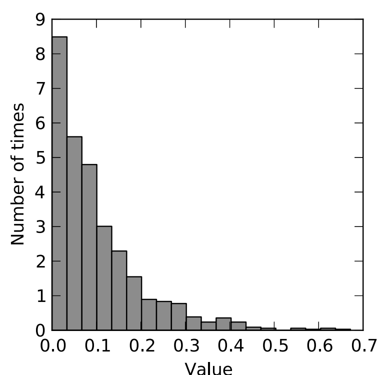


Fig. 8. Simulation results for the exponential distribution.

constructed, we could generate random variables and experiment with different parameters of the system. In the scope of this paper the point of the Monte Carlo experiments is to repeat the trials of our model many times with a view to accumulate and integrate the overall results. The simplest application was described in the previous subsection. If we increase the number of trials, we increase the preciseness of simulation results. Here the learner should carry out a certain number of experiments using this simple model by increasing the number of trials. By increasing the number of dice and the number of trials, the learner will face relatively long calculation times. It could be a good motivation to use parallel calculations. The Python model for multiple dice is presented in Fig. 9 and the result of a simulation is presented in Fig. 10.

```
import pylab
import random

number_of_trials = 150000
number_of_dice=200

## Here we simulate the repeated throwing
## of a number of single six-sided dice
list_of_values = []
for i in range(number_of_trials):
    sum=0
    for j in range(number_of_dice): sum+=random.randint(1,6)
    list_of_values.append(sum)

mean=pylab.mean(list_of_values)
std=pylab.std(list_of_values)
print "Trials =", number_of_trials, "times."
print "Mean =", mean
print "Standard deviation =", std

pylab.hist(list_of_values,20)
pylab.xlabel('Value')
pylab.ylabel('Number of times')
pylab.show()
```

Fig. 9. Python model for the advanced normal distribution.

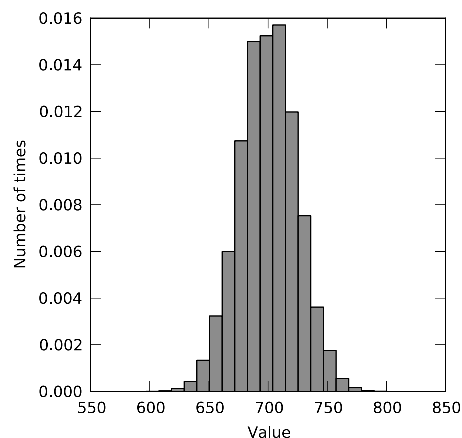


Fig. 10. Simulation results for the advanced normal distribution.

As a next step, a set of more comprehensive problems like modeling of various queueing systems could be introduced for the learner. The brief introduction to classification of queueing systems is presented in the next section of this paper. The learner begins from modeling of M/M/1 system or more complex queueing system. The basic meanings of stochastic processes might be introduced at this step as well. As a possible example, the problem of investigation of the output process could be offered. One can prove that for M/M/1 system the output is again the Poisson process. So the problem of gathering data and plotting the output empirical histogram might be presented.

3. Multiphase queueing systems and stochastic simulation

Below we provide an introductory description of queueing systems that consider modeling and stochastic simulation positions.

3.1. Queueing systems

A simple queueing system consists of one server that provides service for arriving customers. The general scheme of the simple queueing system is presented in Fig. 11.

In general, the queueing system consists of one or more servers that provide service to arriving customers. This could also include one or more servicing phases with one or more servers in each phase. Arriving customers who find all the servers busy join one or more queues in front of the servers. There are many applications that can be modeled as queueing systems, such as manufacturing systems, communication systems, maintenance systems, and so on. An overall queueing system could be characterized by three main components: the arrival process, the service mechanism and the queue discipline. Arrivals may come from one or several limited or unlimited sources.

The arrival process describes how customers arrive to the system. We denote by α_i the interarrival time between the arrivals of the $(i-1)$ and ith customer, the expected inter-arrival time (or mean) by $E(\alpha)$ and the arrival frequency by $\lambda = \frac{1}{E(\alpha)}$.

We denote by s the number of servers in the queueing system. The service mechanism is specified by that number. Each server has its' own queue as well as the probability distribution of customer service time.

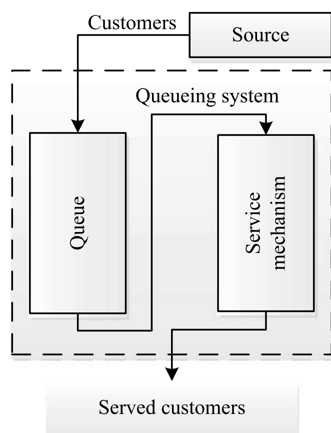


Fig. 11. The simple queueing system.

We denote by s_i the service time of the ith customer, by $E(s)$ the mean service time of a customer and by $\mu = \frac{1}{E(s)}$ the service rate of a server.

The rule that any server uses to choose the next customer from the queue is called the queue discipline of a queueing system. The most used queue disciplines are: Priority – customers are served in the order of their importance; FIFO – customers are served on the first-in first-out basis; LIFO – customers are served on the last-in first-out basis. The extended Kendall classification of queuing systems uses 6 symbols: A/B/s/q/c/p where A is the distribution of intervals between arrivals, B is the distribution of service duration, s is the number of servers, q is the queuing discipline (omitted for FIFO), c is the system capacity (omitted for unlimited queues), p is the number of possible customers (omitted for open systems) [17,37]. For example M/M/1 states for Poisson input, one exponential server, one unlimited FIFO queue, and unlimited customers.

Queueing systems are used for modeling and research in various fields of engineering and science. For example, we could model and study manufacturing or transport systems using the queueing theory. Here the requests for service are considered as customers and the maintenance procedure as a service mechanism. The other examples are: computer systems (terminal requests and server response accordingly), a computer multi-disk memory system (data writing/reading requests, shared disk controller), a trunked radio system (telephone signals, repeaters), local area computer network (requests, channel) [39]. In biology, one could employ a queueing theory to model an enzymatic system (proteins, common enzyme) [8]. In biochemistry one could implement a queueing network model to study the regulatory circuit of the lac operon [1].

3.2. Why multiphase?

We consider a queueing system as multiphase – it consists of more than one server which are joined consequentially, and as unlimited – with unlimited calling population. The interarrival time and the service time are both independent and exponentially distributed variables. The Queue discipline is endless FIFO. A multiphase queueing system naturally maps to a multicore computer topology. As we see in later sections of this report, such a model could be easily programmed, studied and modified. The model also allows a comparative study of different approaches to multiprocessing. The model of the multiphase queueing system is presented in Fig. 12.

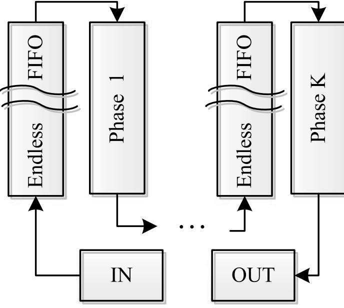


Fig. 12. Multiphase queueing system.

3.3. *Theoretical framework*

In the case of statistical modeling, we always face with the problem of the computer code verification. So it is always an open question if there are any errors in our program or algorithm. The model is not fully analytical and each time we run the program we have different inputs and outputs. So, to verify the correctness of the code or algorithm, a different (from that one we use in the case of fully deterministic input data) approach is needed. To solve this question, we could apply some theoretical results which could be found in the scientific literature. Such results give us a basement for output data analysis and verification as well as for solving the problem of correctness of the modeling results [31,32].

We will investigate the sojourn time of the customer in the multiphase queueing system. We denote by $T_{j,n}$ the sojourn time and by $S_n^{(j)}$ – the service time of the nth customer in the jth phase. We consider α_k as $E(\alpha)$ for the phase number k. We assume that the following conditions are fulfilled:

There exists a constant $\gamma > 0$ such that

$$\sup_{n\geqslant 1} E\left|S_n^{(j)}\right|^{4+\gamma} < \infty, \quad j = 0, 1, 2, \ldots, k \tag{1}$$

and

$$\alpha_k > \alpha_{k-1} > \cdots > \alpha_1 > 0. \tag{2}$$

Theorem. *If conditions* (1) *and* (2) *are fulfilled, then*

$$P\left(\overline{\lim_{n\to\infty}} \frac{T_{j,n} - \alpha_j \cdot n}{\tilde{\sigma} \cdot \alpha(n)} = 1\right) = P\left(\underline{\lim_{n\to\infty}} \frac{T_{j,n} - \alpha_j \cdot n}{\tilde{\sigma} \cdot \alpha(n)} = -1\right) = 1,$$

$$j = 1, 2, \ldots, k \textit{ and } \alpha(n) = \sqrt{2n \ln \ln n}.$$

3.4. *Statistical modeling*

After the model has been constructed, we could arrange a number of experiments with that model. This allows us to investigate certain parameters of the system we study. We could simulate random variables with an expected mean and calculate (using the recurrent equation presented below) the values needed to study. These values will be random as well (we have randomness in the input data of our model – interarrival times and serving times). Afterwards, we can calculate some parameters of such random values (variables) as mean or probability distribution. We call this method as statistical modeling due to the randomness presented in the model. If more reliable results are needed, we must repeat the experiments with our model and then integrate the results i.e. calculate integral characteristics like a mean or a standard deviation. This is called the Monte Carlo method and it was described earlier in this paper.

3.5. *Recurrent equation*

In order to design the modeling algorithm of the previously described queueing system, some additional mathematical constructions should be introduced. Our goal is to calculate and investigate the sojourn time of the customer number n in the multiphase queueing system of k phases. We can prove the next recurrent equation [12]: Let us denote by t_n the time of arrival of the nth customer; by $S_n^{(j)}$ the service time of the nth customer in the jth phase; $\alpha_n = t_n - t_{n-1}$; $j = 1, 2, \ldots, k$; $n = 1, 2, \ldots, N$. The next recurrence equation is valid for the sojourn time $T_{j,n}$ of the nth customer in the jth phase:

$$\begin{aligned} T_{j,n} &= T_{j-1,n} + S_n^{(j)} \\ &\quad + \max(T_{j,n-1} - T_{j-1,n} - \alpha_n, 0); \\ j &= 1, 2, \ldots, k;\ n = 1, 2, \ldots, N; \\ T_{j,0} &= 0, \forall j;\ T_{0,n} = 0, \forall n. \end{aligned}$$

Proposition. *The recurrence equation for calculating of the sojourn time of a customer in a multiphase queueing system.*

Proof. It is true that, if the time $\alpha_n + T_{j-1,n} \geqslant T_{j,n-1}$, the waiting time in the jth phase of the nth customer is 0. In the case $\alpha_n + T_{j-1,n} < T_{j,n-1}$, the waiting time in the jth phase of the nth customer is $\omega_j^n = T_{j,n-1} -$

$T_{j-1,n} - \alpha_n$ and $T_{j,n} = T_{j-1,n} + \omega_j^n + S_n^{(j)}$. Taking into account the above two cases, we finally have the proposition results. □

Now we can start the implementation of the necessary algorithms since all the basic theoretical results have been introduced.

4. Python for multiprocessing

Python as a programming language is very popular among scientists and educators and could be an attractive solution for solving scientifically oriented tasks [3]. Python provides a powerful platform for modeling and simulations including graphical options, a wide amount of mathematical and statistical packages as well as packages for multiprocessing. For time consumable solutions, Python and C code could be combined. All that allows us to implement a powerful modeling platform for statistical modeling and processing of the results of the data. The key Python concepts which are important for modeling are: decorators, coroutines, yield expressions, multiprocessing and queues. A good description of the above is provided by Beazley in his book [2]. Although there are several ways of organizing the inter-process communication, we start from using queues as it is very natural in the context of the queueing systems.

The simple example of the advantage of using multiprocessing in order to increase the efficiency of the programming code is provided below. The learner could proceed with improvements of the provided model by using parallel calculations on supercomputers or computer clusters [28,29]. On the one hand, multiprocessing will allow us to map a multiphase model to the resources of a multicore computer and on the other hand, we could use multiprocessing to perform a number of Monte Carlo trials in parallel. We present these two approaches in the next sections. For motivated learners, a brief introduction to multiprocessing with Python presented below could be provided.

We start from using the mpi4py module. It is important to show to the learner the general idea of how MPI works. It simply copies the provided program to a number of the processor kernels, specified by the user, and integrates the results after using the gather() method. The sample Python code (see Fig. 13) and the results of a simulation (see Fig. 14) are presented.

```
#!/usr/bin/python
import pylab
import random
import numpy as np
from mpi4py import MPI

dice=200
trials=150000

rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()
name = MPI.Get_processor_name()

random.seed(rank)

## Each process - one throwing of a number of six-sided dice

values= np.zeros(trials)

for i in range(trials):
    sum=0
    for j in range(dice): sum+=random.randint(1,6)
    values[i]=sum

data=np.array(MPI.COMM_WORLD.gather(values , root=0))
if rank == 0:
    data=data.flatten()
    mean=pylab.mean(data)
    std=pylab.std(data)

    print "Number of trials =", size*trials, "times."
    print "Mean =", mean
    print "Standard deviation =", std

    pylab.hist(data,20)
    pylab.xlabel('Value')
    pylab.ylabel('Number of times')
    pylab.savefig('multi_dice_mpi.png')
```

Fig. 13. Python model for the advanced normal distribution with MPI.

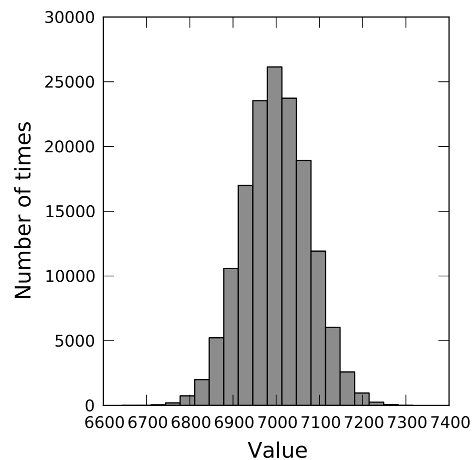


Fig. 14. Normal distribution with MPI.

5. Model-centered educational framework

Multiphase queueing systems provide us a kernel for the development of the relevant model-centered framework. Such a framework includes basic meanings, described in the previous sections, as well as more complex theoretical results and methods. The basic meanings include randomness: random numbers, random number distributions, random numbers generators, Central Limit Theorem; Python programming constructions: decorators, coroutines and yield expressions. More complex results include theoretical facts such as the sojourn time of the customer, the recurrent equation for calculating the sojourn time, stochastic simulation methods and multiprocessing techniques. In Fig. 15, we provide a general scheme of the described educational framework.

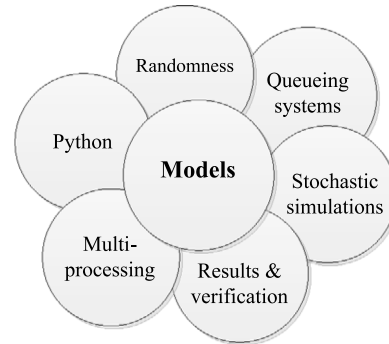


Fig. 15. Model centered educational framework.

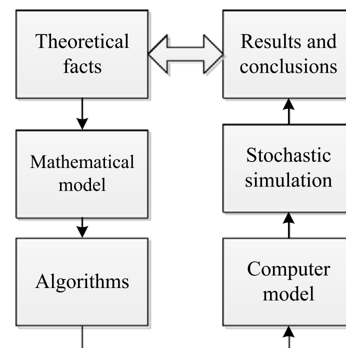


Fig. 16. The scope of scientific research.

All these theoretical and programming structures allow the learner to carry out experiments with different models of multiphase queueing systems. The aim for such experimentations is twofold. First of all, it enables the learner to understand the next sequence, which is important in any scientific research: theoretical facts to be studied, mathematical model, programming constructions, computer model, stochastic simulation and observation of simulation results. That will give the whole picture of the scope of the general scientific research to the learner (see Fig. 16).

Then it forces a deep understanding of stochastic simulations and basic programming constructions like multiprocessing and parallel programming. Such competences are of primary importance in the field of scientific computing.

5.1. Experiments with models

In this section, we provide three computer models of the multiphase queueing system. Each of these models is rather different by its philosophy and key features. Although the aim of each of these models is to statistically model and investigate the main parameters of the multiphase system, the ideas which stay behind the scene of these models, are completely different. A comparison of these basic ideas will help the learner to understand the main fundamentals that lie behind the parallel calculations, multiprocessing statistical modeling and simulation.

The first model presented by us is based on the real time recordings and we call it an imitative model. It uses the Python multiprocessing module. The precision of this model depends on the precision and resolution of the *time*() method. It could be rather low in the case of various general-purpose operation systems and rather high in the case of the real time operation systems (RTOS). The learner could modify this model using the earlier presented recurrent equation (for the sojourn time calculations) and compare the results in both cases.

The next model calculates the sojourn time of the customer and is based on stochastic simulations. The model does not use multiprocessing directly. It emulates multiprocessing by using Python yield expressions.

The last model presented here uses the Python MPI mpi4py module. Now we use real MPI techniques for statistical modeling and could enhance Monte Carlo simulations by additional trials.

In general, the task for the learner is to provide a series of experiments with the presented models and to obtain the experimental proof of the law of the iterated logarithm for the sojourn time of the customer in the case of the multiphase queueing system.

5.2. The imitative model based on multiprocessing of services

Below we present the imitative model. The main issue to study is the difference between the imitative and

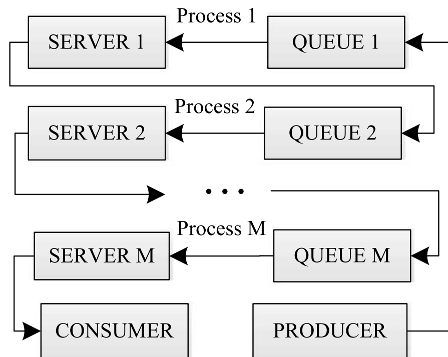


Fig. 17. The imitative model.

statistical models. Another important question is the correctness or precision of the imitative model. It is also important to solve the question of verification of the presented model. The learner could study and compare modeling results depending on various modeling parameters such as interarrival and servicing frequencies, number of customers and number of services. The general schema of the model is presented in Fig. 17.

The programming code consists of two main parts. The first one is directly intended for calculations and the next one is for plotting of the results. The module for calculations contains three main functions: producer() – for producing customers and putting them to the first queue; server() – for serving the customers; consumer() – for finalizing the results. This programming model is based on real simulations and uses no mathematical equations for calculations. Its precision depends on the precision of the Python timing module and generally varies depending on the operating system. Servers are distributed between various processes inside the multiprocessing system. The computer code for implementing of the above model is presented in Fig. 18.

Questions to be studied:

- How global variables are shared between processes?
- How the processes, associated with different servers, will terminate?
- How the informational flow between various processes is transferred?
- What about the correctness of the model?
- What about the efficiency of the model. How long does it takes for different processes to exchange information?

Now we can print the results using the Python matplotlib module and we can visually analyze the results after the plot is prepared. We can see (see Fig. 19) that the model needs further improvements. So we can proceed with a more powerful model.

5.3. *The single process statistical model*

The main features of the statistical model are as follows next: now we use the recurrent equation for exact calculations of the customer's sojourn time; we process all the data in a single process using Python specific coroutine functions; we proceed with a definite number of Monte Carlo simulations for a better validity of the calculations. This model gives us "exact" calculations of the sojourn time. The general schema of the model is presented in Fig. 20. The learner could study the differences between the imitative and statistical models.

The computer code for implementing of the above model is presented in Fig. 21. Simulation results are presented in Fig. 22.

5.4. *Statistical model strengthened by MPI*

The next step is to strengthen our model using the Python MPI module – mpi4py. It allows us to proceed with more Monte Carlo simulations and to a use computer cluster for running and testing the model. The next step could be a further improvement of the model by using the C programming language, "real" MPI or SWIG (Simplified Wrapper and Interface Generator) technology for Python. This model is almost identical to the previous model with the only difference that it uses mpi4py for multiprocessing and integrating the results (see Fig. 23).

In addition to the previous model, several additional modules need to be imported. The *print_results*() function also needs to be rewritten, because we now have more trials. We should also rewrite the main part of the program. In Fig. 24 we present only that part of the computer code which differs from the code of the previous model. Simulation results are presented in Fig. 25.

6. Conclusions

In this paper, a number of models for model-centered learning are provided. These models enable the learner to conduct a series of experiments and enhance the understanding of the Scientific Computing

```
import multiprocessing
import time
import random
import numpy as np

def server(input_q,next_q,i):
    while True:
        item = input_q.get()
        if i==0:item.st=time.time()  ## start recording time
                                              ## (first phase)
        time.sleep(random.expovariate(glambda[i]))
##stop recording time (last phase)
        if i==M-1:item.st=time.time()-item.st
        next_q.put(item)
        input_q.task_done()
    print("Server%d stop" % i) ##will be never printed why?

def producer(sequence,output_q):
    for item in sequence:
        time.sleep(random.expovariate(glambda[0]))
        output_q.put(item)

def consumer(input_q):
    "Finalizing procedures"
     ## start recording processing time
    ptime=time.time()
    in_seq=[]
    while True:
        item = input_q.get()
        in_seq+=[item]
        input_q.task_done()
        if item.cid == N-1:
            break
    print_results(in_seq)
    print("END")
    print("Processing time sec. %d" %(time.time()-ptime))
    ## stop recording processing time
    print("CPU used %d" %(multiprocessing.cpu_count()))

def print_results(in_seq):
    "Output rezults"
    f=open("out.txt","w")
    f.write("%d\n" % N)
    for t in range(M):
        f.write("%d%s" % (glambda[t],","))
    f.write("%d\n" % glambda[M])

    for t in range(N-1):
        f.write("%f%s" % (in_seq[t].st,","))
    f.write("%f\n" % (in_seq[N-1].st))
    f.close()

class Client(object):
    "Class client"
    def __init__(self,cid,st):
        self.cid=cid  ## customer id
        self.st=st ## sojourn time of the customer

###GLOBALS
N=100  ## total number of customers arrived
M=5   ## number of servers
### glambda - arrival + servicing frequency
### = customers/per time unit
glambda=np.array([30000]+[i for i in np.linspace(25000,5000,M)])

 ###START
if __name__ == "__main__":
    all_clients=[Client(num,0) for num in range(0,N)]
    q=[multiprocessing.JoinableQueue() for i in range(M+1)]

    for i in range(M):
        serv = multiprocessing.Process(target=server,args=(q[i],q[i+1],i))
        serv.daemon=True
        serv.start()

    cons = multiprocessing.Process(target=consumer,args=(q[M],))
    cons.start()

    ### start 'produsing' customers
    producer(all_clients,q[0])

    for i in q: i.join()
```

Fig. 18. Python code for the imitative model based on multiprocessing of services.

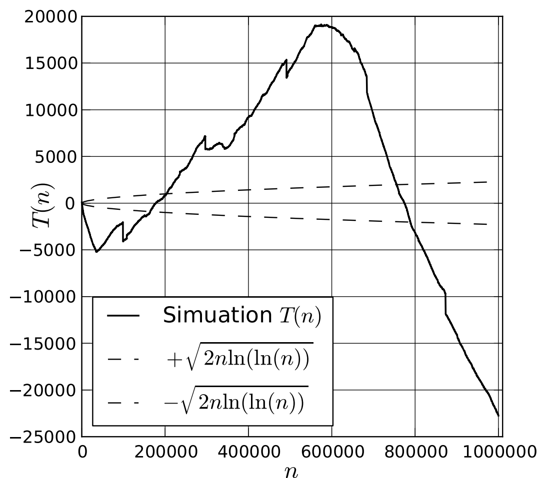


Fig. 19. Simulation results for the imitative model.

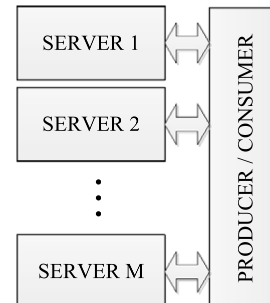


Fig. 20. The single-process statistical model.

```
#!/usr/bin/python
import random
import time
import numpy as np
from numpy import linspace

def coroutine(func):
    def start(*args,**kwargs):
        g = func(*args,**kwargs)
        g.next()
        return g
    return start

def print_header():
    "Output rezults - header"
    f=open("out.txt","w")
    f.write("%d\n" % N)
    ##number of points in printing template
    f.write("%d\n" % TMPN)
    for t in range(M):
        f.write("%d%s" % (glambda[t],","))
    f.write("%d\n" % glambda[M])
    f.close()

def print_results(in_seq):
    "Output rezults"
    f=open("out.txt","a")
    k=0
    for i in range(N-2):
        if in_seq[i].cid==template[k]:
            f.write("%f%s" % (in_seq[i].st,","))
            k+=1
    f.write("%f\n" % (in_seq[N-1].st))
    f.close()

@coroutine
def server(i):

    ST=0  ##sojourn time for the previous client
    item=None
    while True:
        item = (yield item)  ##get item
        if  item == None:    ##new Monte Carlo iteration
            ST=0
            continue
        waiting_time=max(0.0,ST-item.st-item.tau)
item.st+=random.expovariate(glambda[i+1])+waiting_time
ST=item.st

def producer():
    results=[]
    i=0
    while True:
        if i == N: break
        c=Client(i,0.,0.)
        if i!=0: c.tau=random.expovariate(glambda[0])
        i+=1
        for s in p: c=s.send(c)
        results+=[c]
    for s in p: c=s.send(None)  ##final signal
    return results

class Client(object):
    def __init__(self,cid,st,tau):
        self.cid=cid
        self.st=st
        self.tau=tau
    def params(self):
        return (self.cid,self.st,self.tau)

stt=time.time()

N=1000000  ## Clients
M=5        ## Servers

## Input/sevice frequency
glambda= [30000]+[i for i in linspace(25000,5000,M)]
MKS=20   ## Monte Carlo simulation results

## Number of points in the printing template
TMPN=N/10000

##printing template
template= map(int,linspace(0,N-1,TMPN))

print_header()

p=[]
for i in range(M):p +=[server(i)]
for i in range(MKS):
    print_results(producer())
    print("Step=%d" % i)

sys.stdout.write("Processing time:%d\n" % int(time.time()-stt))
```

Fig. 21. Python code for the single process statistical model.

discipline. There are several difficulty levels of the presented models and experiments with such models. The first level is the basic one. It provides an introduction to randomness and also enables primary understanding of the scope of the scientific research. The next one is more sophisticated and enables a deep understanding of parallel programming and stochastic simulations. The relevant theoretical knowledge is provided on demand and as a supporting material for the learner's activities. All that provides a constructivist framework for the model-centered introduction to the Scientific Computing. Finally, we would like to provide recommendations for further study and improvements of the models.

6.1. *Linearity of the model and statistical parameters of the queueing system*

The model of the multiphase queueing system, provided in this paper, is not linear [12]. It is obvious from the recurrent equation since it contains a nonlinear mathematical function *max*. If we want to obtain correct modeling results, especially in the case of calculating the statistical parameters of the queueing system, we must use a partially linear model for calculations. This is particularly important for non-heavy traffic systems, as in this case we could make rather great mistakes in calculations.

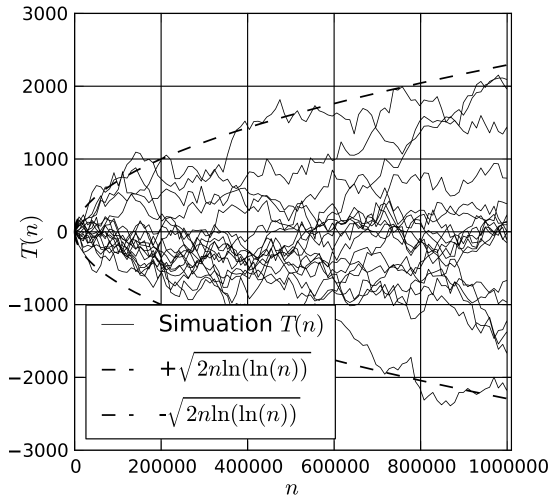


Fig. 22. Simulation results for the single process statistical model.

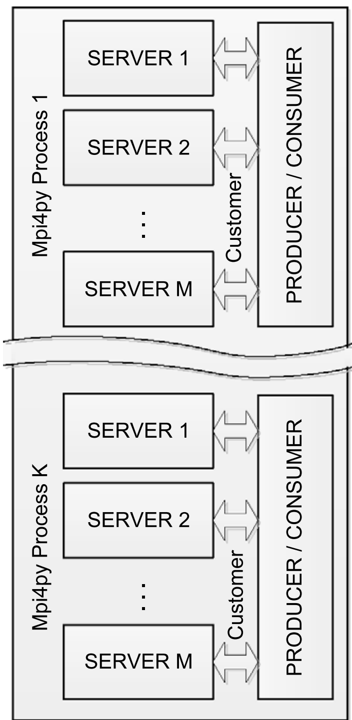


Fig. 23. The MPI statistical model.

6.2. Extensions of Python modules and parallel programming with C

For the skillful learners, it could be interesting to proceed with improvements of the efficiency of the programming code. That could be done by extending Python modules with C implemented functions using the SWIG technology. Learners could improve the code and speed-up calculations using Cython or C programming languages, "real" MPI technology and HTC (High Throughput Computing) cluster possibilities [5, 28,29].

```
……………..
import sys
from mpi4py import MPI
……………..
def print_results(in_seq):
   "Output rezults"
   f=open("out.txt","a")
   for m in range(int(size)):
      for j in range(MKS):
         for i in range(TMPN-1):
            f.write("%f%s" % (in_seq[m][i+j*TMPN].st,","))
         f.write("%f\n" % (in_seq[m][(TMPN-1)+j*TMPN].st))
   f.close()
………………
stt=time.time()   #start time for the process

rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()
name = MPI.Get_processor_name()

N=10**3  ## Clients
M=5      ## Servers
## Input/sevice frequency
glambda= [30000]+[i for i in linspace(25000,5000,M)]
## Number of Monte-Carlo simulations for this  particuar process
MKS=20
TMPN=200  ## Number of points in printing template
template= map(int,linspace(0,N-1,TMPN))  ## points for printing
p=[]
results=[]          ## this process results
total_results=[]  ## overall results
for i in range(M):p +=[server(i)]
for i in range(MKS):results+=producer()

total_results=MPI.COMM_WORLD.gather(results,0)
random.seed(rank)

if rank == 0:
   print_header()
   print_results(total_results)
   sys.stdout.write("Processing time: %d\n" % int(time.time()-stt))
```

Fig. 24. Python code for statistical model strengthened by MPI.

6.3. Efficiency of the programming solutions and further work

In this section, the learner could study the efficiency of various programming solutions. This topic is particularly important for any programming model, which is based on parallel calculations. In this case, learners could study the effectiveness of different programming models and could try to improve algorithms step by step. The key point here is to investigate the ratio

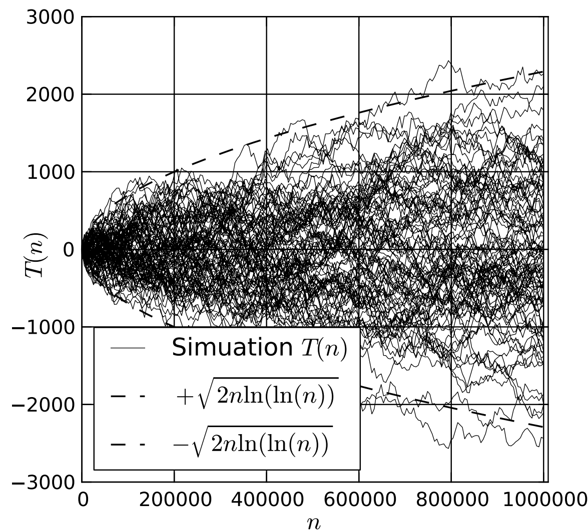


Fig. 25. Simulation results for the MPI statistical model.

of the amount of information flow and calculations for different programming processes. Such a ratio is important when constructing the most effective programming model for parallel calculations. Another interesting topic is to study possible mappings of the algorithm structure to the HTC cluster structure.

As a further task for investigations, the authors consider studies of queueing networks to be introduced to the learner, modeled, and analyzed. The comparatively complex nature of queueing networks and variety of applications requires more comprehensive programming techniques to be involved. This provides a good basic platform for introduction of such general programming concepts like inheritance, encapsulation, and polymorphism. On the other hand, the basic theoretical computer science constructions needed to be introduced as well. Besides all these, the modeling and statistics simulation of queueing networks requires more advanced probability topics to be presented, more computational resources to be occupied and provide a real scientific computing environment and good motivation for the advanced learner.

References

[1] A. Arazi, E. Ben-Jacob and U. Yechiali, Bridging genetic networks and queueing theory, *Physica A: Statistical Mechanics and Its Applications* **332** (2004), 585–616.

[2] D.M. Beazley, *Python Essential Reference*, Addison-Wesley Professional, 2009.

[3] J. Bernard, Use Python for scientific computing, *Linux Journal* **175** (2008), 7.

[4] U.N. Bhat, *An Introduction to Queueing Theory Modeling and Analysis in Applications*, Birkhäuser, Boston, MA, 2008.

[5] K.J. Bogacev, *Basics of Parallel Programming*, Binom, Moscow, 2003.

[6] R.N. Caine and G. Caine, *Making Connections: Teaching and the Human Brain*, Association for Supervision and Curriculum Development, Alexandria, 1991.

[7] J. Clement and M.A. Rea, *Model Based Learning and Instruction in Science*, Springer, The Netherlands, 2008.

[8] N.A. Cookson, W.H. Mather, T. Danino, O. Mondragón-Palomino, R.J. Williams, L.S. Tsimring and J. Hasty, Queueing up for enzymatic processing: correlated signaling through coupled degradation, *Molecular Systems Biology* **7** (2011), 1.

[9] A.S. Gibbons, Model-centered instruction, *Journal of Structural Learning and Intelligent Systems* **4** (2001), 511–540.

[10] M.T. Heath, *Scientific Computing an Introductory Survey*, McGraw-Hill, New York, 1997.

[11] A. Hellander, *Stochastic Simulation and Monte Carlo Methods*, 2009.

[12] G.I. Ivcenko, V.A. Kastanov and I.N. Kovalenko, *Queuing System Theory*, Visshaja Shkola, Moscow, 1982.

[13] Z.L. Joel, N.W. Wei, J. Louis and T.S. Chuan, Discrete–event simulation of queuing systems, in: *Sixth Youth Science Conference*, Singapore Ministry of Education, Singapore, 2000, pp. 1–5.

[14] E. Jones, Introduction to scientific computing with Python, in: *SciPy*, California Institute of Technology, Pasadena, CA, 2007, p. 333.

[15] M. Joubert and P. Andrews, Research and developments in probability education internationally, in: *British Congress for Mathematics Education*, 2010, p. 41.

[16] G.E. Karniadakis and R.M. Kyrby, *Parallel Scientific Computing in C++ and MPI. A Seamless Approach to Parallel Algorithms and Their Implementation*, Cambridge Univ. Press, 2003.

[17] D.G. Kendall, Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain, *The Annals of Mathematical Statistics* **1** (1953), 338–354.

[18] M.S. Khine and I.M. Saleh, Models and modeling, cognitive tools for scientific enquiry, in: *Models and Modeling in Science Education*, Springer, 2011, p. 290.

[19] T. Kiesling and T. Krieger, Efficient parallel queuing system simulation, in: *The 38th Conference on Winter Simulation*, Winter Simulation Conference, 2006, pp. 1020–1027.

[20] J. Kiusalaas, *Numerical Methods in Engineering with Python*, Cambridge Univ. Press, 2010.

[21] A. Kumar, *Python for Education. Learning Maths & Science Using Python and Writing Them in LATEX*, Inter University Accelerator Centre, New Delhi, 2010.

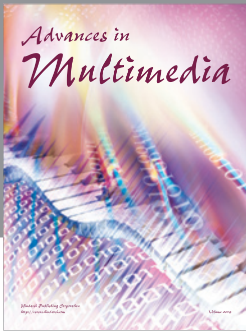
[22] H.P. Langtangen, *Python Scripting for Computational Science*, Springer-Verlag, Berlin, 2009.

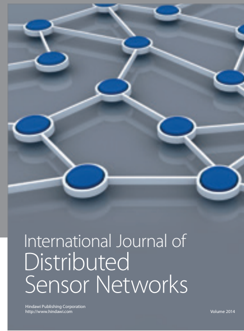
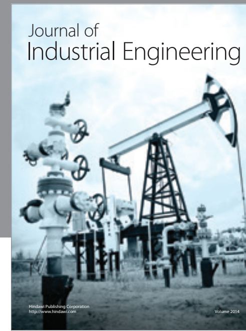
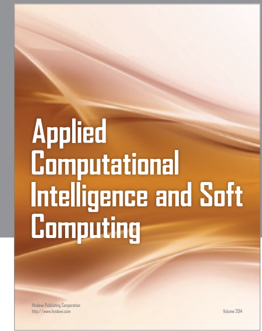
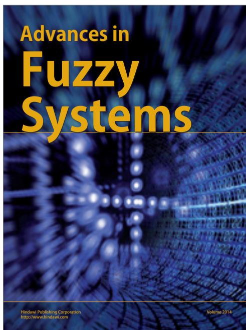
[23] H.P. Langtangen, *A Primer on Scientific Programming with Python*, Springer-Verlag, Berlin, 2011.

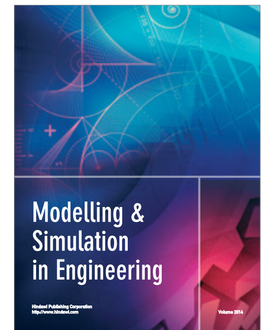
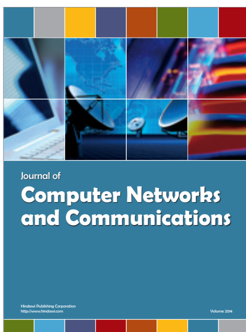
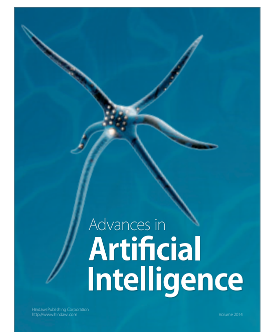
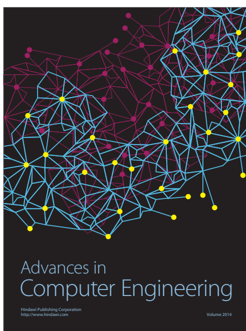
[24] H.P. Langtangen, Experience with using Python as a primary language for teaching scientific computing at the University of Oslo, University of Oslo, 2012.

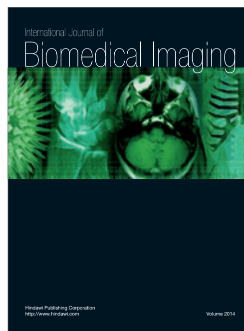
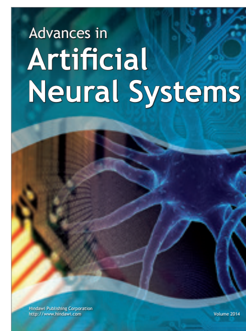
[25] R. Lehrer and L. Schauble, Cultivating model-based reasoning in science education, in: *The Cambridge Handbook of the Learning Sciences*, Cambridge Univ. Press, 2005, pp. 371–388.

[26] G. Levy, An introduction to quasi-random numbers, in: *Numerical Algorithms*, Group, 2012.
[27] J.S. Liu, *Monte Carlo Strategies in Scientific Computing*, Harvard Univ., 2001.
[28] V.E. Malishkin and V.D. Korneev, *Parallel Programming of Multicomputers*, Novosibirsk Technical Univ., Novosibirsk, 2006.
[29] N. Matloff, *Programming on Parallel Machines: GPU, Multicore, Clusters and More*, University of California, 2012.
[30] M. Milrad, J.M. Spector and P.I. Davidsen, Model facilitated learning, in: *Instructional Design, Development and Evaluation*, Syracuse Univ. Press, 2003.
[31] S. Minkevičius, On the law of the iterated logarithm in multiphase queueing systems, *Informatica* **II** (1997), 367–376.
[32] S. Minkevičius and V. Dolgopolovas, Analysis of the law of the iterated logarithm for the idle time of a customer in multiphase queues, *Int. J. Pure Appl. Math.* **66** (2011), 183–190.
[33] Model-Centered Learning, Pathways to mathematical understanding using GeoGebra, in: *Modeling and Simulations for Learning and Instruction*, Sense Publishers, The Netherlands, 2011.
[34] C.R. Myers and J.P. Sethna, Python for education: Computational methods for nonlinear systems, *Computing in Science & Engineering* **9** (2007), 75–79.
[35] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*, SIAM, 1992.
[36] F.B. Nilsen, Queuing systems: Modeling, analysis and simulation, Department of Informatics, University of Oslo, Oslo, 1998.
[37] R.P. Sen, *Operations Research: Algorithms and Applications*, PHI Learning, 2010.
[38] F. Stajano, Python in education: Raising a generation of native speakers, in: *8th International Python Conference*, Washington, DC, 2000, pp. 1–5.
[39] J. Sztrik, Finite-source queueing systems and their applications, *Formal Methods in Computing* **1** (2001), 7–10.
[40] L. Xue, Modeling and simulation in scientific computing education, in: *International Conference on Scalable Computing and Communications*, 2009, pp. 577–580.

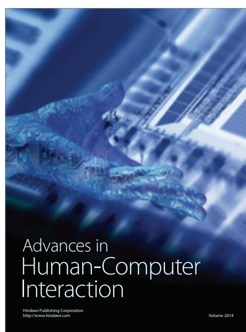
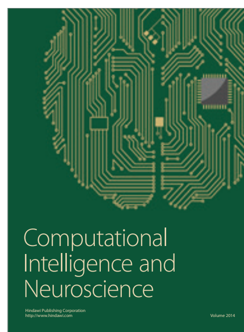

Advances in
Multimedia

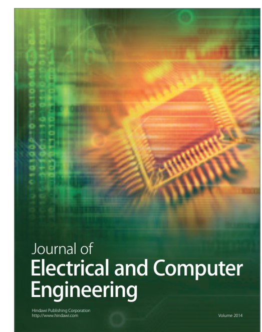
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

Hindawi
Submit your manuscripts at
http://www.hindawi.com

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering