ПРОГРАММИРОВАНИЕ, 2020, № 2, с. 30–37

КОМПЬЮТЕРНАЯ АЛГЕБРА

УДК 004.422.8

ПРИМЕР МОДУЛЬНОГО РАСШИРЕНИЯ СИСТЕМЫ КОМПЬЮТЕРНОЙ АЛГЕБРЫ

© 2020 г. М. Н. Геворкян[a,*], А. В. Королькова[a,], Д. С. Кулябов[a,b,***], Л. А. Севастьянов[a,c,****]**

[a] *Кафедра прикладной информатики и теории вероятностей, Российский университет дружбы народов, ул. Миклухо-Маклая, д. 6, 117198 Москва, Россия*

[b] *Лаборатория информационных технологий, Объединенный институт ядерных исследований, ул. Жолио-Кюри 6, 141980 Дубна, Московская область, Россия*

[c] *Лаборатория теоретической физики, Объединенный институт ядерных исследований, ул. Жолио-Кюри 6, 141980 Дубна, Московская область, Россия*

**E-mail: gevorkyan-mn@rudn.ru*

***E-mail: korolkova-av@rudn.ru*

****E-mail: kulyabov-ds@rudn.ru*

*****E-mail: sevastianov-la@rudn.ru*

Поступила в редакцию 08.09.2019 г.
После доработки 19.10.2019 г.
Принята к публикации 19.10.2019 г.

Системы компьютерной алгебры представляют из себя сложные программные комплексы, охватывающие широкий спектр научных и практических проблем. Однако абсолютная полнота недостижима. И зачастую возникает задача создания пользовательского расширения существующей системы компьютерной алгебры. При этом следует учитывать расширяемость самой системы. В статье рассматривается технология расширения системы компьютерной алгебры SymPy низкоуровневым модулем, реализующим генератор случайных чисел.

DOI: 10.31857/S0132347420020065

1. ВВЕДЕНИЕ

Система компьютерной алгебры SymPy [1] является по своей сути модулем, написанным на языке Python [2, 3], поэтому для расширения функциональности SymPy достаточно написать функцию или модуль на самом Python.

Хотя язык Python является универсальным языком программирования и на нем можно реализовать любые алгоритмы, однако ему присущ существенный недостаток – малая производительность. Падение быстродействия особенно заметно в случае если в алгоритме присутствуют циклы. Данная проблема обусловлена динамической природой языка из-за которой даже элементарные типы данных, такие как `int` и `float` реализованы в стандартном интерпретаторе *cpython* в виде составных структур данных.

Язык Python хорошо подходит для прототипирования приложения. Также язык Python является в некотором роде клеем – он хорошо подходит для связывания разных библиотек вместе [4]. Но попытка создать на основе этого языка большую быструю программу скорее всего обречена на провал. Причина, по которой он так успешно справляется с научными и инженерными задачами состоит в том, что Python использует низкоуровневый интерфейс к библиотекам, написанным на более вычислительно эффективных языках программирования. Таким образом создается впечатление, что Python работает так же быстро, как и код, написанный, например, на C++. Побочным эффектом этого является то, что для практического программирования на Python необходимо также программировать и на языках более низкого уровня. Если для задачи достаточно использования стандартных библиотек, то ничего кроме Python может и не понадобится. Однако, если необходимо добавить новую функциональность, то следует использовать более низкоуровневые языки программирования.

В данной статье рассмотрено использование модуля языка Python `ctypes` для интеграции C-функций в Python-программу. В качестве примера рассматривается библиотека, реализующая

генератор случайных чисел. Этот пример интересен тем, что он является достаточно ресурсоемким и его нецелесообразно делать на чистом Python.

Следует заметить, что идея использования компилируемого языка со статической типизацией для повышения производительности отдельных элементов программы, написанной на интерпретируемом языке с динамической типизацией, не является новой [5, 6]. Напротив, исторически сложилось множество технологических подходов, позволяющих осуществить такое расширение. Такое многообразие подходов затрудняет доступ начинающих в эту область. В нашей статье дается краткий обзор подходов, позволяющих повысить быстродействие Python-программ и краткая характеристика каждого из них.

Основной упор делается на использование встроенного модуля `ctypes` для непосредственного вызова С-функций из Python-программ. Обосновывается выбор именно данного модуля. Изложение носит практический характер и затрагивает вопросы создания библиотеки на языке Си, ее компиляции для дальнейшего использования с `ctypes` под различные платформы. Данная часть статьи может использоваться как введение в возможности `ctypes` для начинающих. В последней части статьи описывается созданный нами модуль, использующий `ctypes` для вызова функций из библиотеки на языке С, которая реализует ряд генераторов псевдослучайных чисел. Приведены тесты для сравнения предложенного модуля генератора псевдослучайных чисел с имеющимися генераторами из модуля *random* и библиотеки *NumPy*.

2. СПОСОБЫ ПОВЫШЕНИЯ БЫСТРОДЕЙСТВИЯ SYMPY/PYTHON ПРОГРАММ

Перечислим основные средства, которые используются в настоящее время для повышения быстродействия Python-программ.

• Использование сторонних библиотек, таких как *NumPy* [7, 8] и *SciPy* [9], в которых ресурсоемкие алгоритмы реализованы на языках C/C++ и Fortran, а сам Python используется как язык-связка для предоставления удобного программного интерфейса.

• Оптимизирующий статический компилятор *Cython* [10, 11] который позволяет транслировать Python-код в код на C/C++. При этом сама программа пишется на специальном диалекте Python, который позволяет применять статическую типизацию в критичных для производительности участках кода.

• Проект *Numba* [12], представляющий собой JIT-компилятор Python кода. Numba позволяет писать программу на чистом Python, используя декораторы для функций и циклов, производительность которых необходимо повысить.

• Модуль `ctypes` [13] из стандартной библиотеки интерпретатора Cpython, который позволяет непосредственно вызвать С-функции, из статических или разделяемых (shared) библиотек, как обычные Python-функции.

Заметим, что ни одно из перечисленных средств не является универсальным. Специализированные библиотеки *NumPy* и *SciPy* нацелены в основном на научные и инженерные вычислительные задачи, поэтому реализуемый ими набор алгоритмов хоть и обширен, но ограничен рамками этой специализации. JIT-компилятор *Numba* дает существенный прирост скорости, однако проект пока находится на стадии разработки и его функциональные возможности ограничены стандартными сценариями применения.

Статический компилятор *Cython* на сегодняшний день является одним из самых часто используемых средств для повышения производительности. Его используют многие библиотеки, в том числе *NumPy* и *SciPy* для повышения производительности и интеграции библиотек на C/C++.

В данной работе мы используем модуль `ctypes`, поскольку для наших задач он имеет ряд преимуществ над *Cython*:

• это стандартный модуль Cpython, тогда как *Cython* необходимо устанавливать отдельно;

• не смешиваются несколько диалектов языка Python;

• `ctypes` особенно полезен, если необходимая функциональность уже реализована на С. В этом случае подготовка вызова функций крайне проста и занимает буквально несколько строк кода. Также его разумно применять в случае, если реализуемые С-функции просты, но активно используют циклы и работу с примитивными типами данных.

Таким образом, при использовании `ctypes` работа над программой делится на два этапа. На первом этапе программист реализует функции на языке С, компилирует и собирает из них статическую или разделяемую (динамическую) библиотеку. Второй шаг – отдельная реализация ряда функций-оберток уже на языке Python. Данные функции-обертки по сути представляют собой интерфейс для удобного вызова уже реализованных С-функций из Python-программ. Отметим, что функциональность вызова С-функций

реализуется с помощью модуля ctypes, входящего в стандартную библиотеку интерпретатора CPython.

3. МОДУЛЬ CTYPES

В данном разделе описан полный цикл создания библиотеки на языке Си и ее использование совместно с ctypes. Официальная документация ctypes приводит примеры использования функций из данного модуля, однако в ней не затрагивается процесс создания библиотеки на языке С.

Использование модуля ctypes начинается с загрузки файла библиотеки, поэтому прежде чем переходить к описанию базовой функциональности ctypes, приведем пример сборки статической и динамической (разделяемой) библиотек на примере компилятора языка С из набора компиляторов *gcc*. Авторы использовали gcc версии 8.3.0 под операционной системой Gnu Linux, дистрибутив Ubuntu 19.04.

3.1. Компиляция С-функций и сборка библиотеки

Код типичной библиотеки на языке С состоит из набора файлов с исходным кодом (crc_01.c, crc_02.c и т.д.) и ряда заголовочных файлов (header_01.h и т.д.). Общепринятой практикой [14] является размещение всех файлов с исходным кодом в поддиректории src проекта, а заголовочных файлов в поддиректории include.

Для компиляции исходных файлов используются следующие ключи компилятора.

• -c – позволяет создать объектный файл, без сборки всей программы или библиотеки.

• -Wall – компилятор будет распечатывать сообщения не только о синтаксических ошибках, но и предупреждения, которые потенциально могут привести к некорректной работе программы.

• -Werror – все предупреждения будут интерпретироваться компилятором как ошибки.

• -fPIC – указывает компилятору о необходимости транслировать программу в позиционно-независимый машинный код (position-independent code – PIC), где все переходы осуществляются только по относительным адресам. Этот флаг важен, так как библиотека потенциально может быть загружена в любом месте программы.

• -I./include – указывает компилятору, что файлы *заголовков* следует искать в локальной директории include нашего проекта.

• -L./lib – указывает компилятору, что файлы *библиотек* следует искать в локальной директории lib нашего проекта. Важно соблюдать последовательность и указывать флаг -L только после флага -I.

• После отладки программы можно также добавить флаг оптимизации -O2 или -O3, помня, однако, что агрессивная оптимизация в некоторых случаях может привести к некорректной работе функций.

Все перечисленные флаги сохраняем в переменную окружения CFLAGS и для компиляции файла с исходным кодом в объектный файл для каждого файла выполняется следующая команда:

```
gcc -c $CFLAGS src/src_01.c -o
↪lib/obj_01.o
```

После того, как будут созданы все объектные файлы, их можно запаковать в статическую библиотеку с помощью утилиты ar, выполнив следующую команду:

```
ar crs libmy.a lib/obj_01.o lib/obj_02.o
```

Опции crs говорят о том, что нужно создать архив с заменой файлов, если таковые уже в нем есть и дополнительно создать индекс. После успешного выполнения команды получим файл статической библиотеки libmy.a, который можно будет использовать для подключения средствами ctypes.

Для создания статической библиотеки в среде Windows следует использовать опцию -Wl, которая позволит передать дополнительные опции компоновщику (linker) и указать с помощью опции компоновщика --out-implib путь к файлу статической библиотеки, которую необходимо создать.

```
gcc -shared lib/obj_01.o lib/obj_02.o
↪libmy.so -o bin/libmy.dll
↪-Wl,-out-implib,lib/win/libmy.a
```

При необходимости, можно создать не статическую, а разделяемую библиотеку:

```
gcc -shared lib/obj_01.o lib/obj_02.o
↪libmy.so
```

В результате получим файл разделяемой библиотеки libmy.so. Та же команда позволяет получить динамическую библиотеку и в системе Windows. Следует лишь указать расширение файла как dll вместо so.

3.2. Загрузка библиотеки в Python-программу

Предполагая, что мы успешно скомпилировали и собрали библиотеку libmy.so, опишем процедуру ее импорта в Python-программу. Предполагаем, что файл библиотеки будет находиться в той же директории, что и наша Python-программа. Разберем следующий фрагмент кода.

```
import ctypes
import sys
import os
path = os.path.dirname(os.path.realpath(
↪ __file__ ))
if sys.platform.startswith('win'):
    clib = ctypes.CDLL(os.path.join(path,
    ↪ 'libmy.dll'))
else:
    clib = ctypes.CDLL(os.path.join(path,
    ↪ 'libmy.so'))
```

Вначале необходимо загрузить модуль `ctypes` и ряд дополнительных модулей. Далее получаем абсолютный путь до директории с программой. Затем определяем тип операционной системы и в зависимости от этого загружаем файл `.dll` или `.so`.

Стоит отметить, что загрузка файла библиотеки по абсолютному пути обязательна в том случае, если мы организуем нашу Python-программу в виде модуля и хотим хранить файл библиотеки внутри директории модуля.

3.3. Вызов функций из библиотеки

После импорта все функции библиотеки будут доступны для вызова в виде атрибутов объекта `clib`. Пусть, например, в библиотеке `libmy` присутствует следующая функция:

```
uint64_t uint64_var(uint64_t var) {
    uint64_t i = 9223372036854775807llu;
    printf("Function uint64_var, arg
    ↪ uint64 var = %llu\n", i);
    return i;
}
```

Для ее вызова из Python-программы можно использовать следующий код:

```
clib.uint64_var.argtypes =
↪ [ctypes.c_uint64]
clib.uint64_var.restype =
ctypes.c_uint64
res = clib.uint64_var(0)
```

Перед вызовом функции мы указали тип аргумента используя список из одного элемента, так как аргумент единственный. Далее указывается тип возвращаемого значения, после чего можно вызвать требуемую функцию. В ctypes определены все стандартные типы языка C и вызов любой простой функции, принимающей и возвращающей аргументы базовых типов, укладывается в вышеприведенные три строки кода.

Рассмотрим чуть более сложный пример, когда аргумент передается в функцию по указателю. Пусть имеется следующая C-функция:

```
void change_var(double* var) {
    *var = 2.0;
}
```

Следующий код показывает способ вызвать эту функцию с помощью `ctypes`:

```
x = ctypes.c_double(1.0)
print(f"x = {x.value}")
clib.change_var(ctypes.byref(x))
print(f"x = {x.value}")
```

Здесь мы вначале с помощью конструктора `c_double` присвоили значение переменной `x`, а затем передали ее в виде аргумента функции `change_var`, указав дополнительно с помощью `byref`, что аргумент передается по ссылке. Так как функция не возвращает никаких значений, то не нужно указывать `restype`, а так как мы передали в качестве аргумента переменную уже известного типа, то указывать тип аргумента тоже не пришлось.

Наконец рассмотрим вызов функции, принимающей в качестве аргумента массив:

```
double avg_value(long long int array[],
↪ size_t len) {
   double avg = 0.0;
   for (size_t i = 0; i < len; ++i) {
        avg += (double) array[i] /
        ↪ (double) len;
        array[i] = 0.0;
   }
   return avg;
}
```

Функция `avg_value` принимает в качестве первого аргумента массив, а в качестве второго целое число – размер массива. Для удобного вызова этой функции из Python-кода можно написать следующую функцию-обертку.

```
def avg_value(l: list) -> float:
    """Обертка для avg_value"""
    clib.avg_value.restype =
    ↪ ctypes.c_longdouble
    A = (ctypes.c_longlong * len(l))(*l)
    n = ctypes.c_size_t(len(l))
    return clib.avg_value(A, n)
```

Вначале определяется возвращаемый тип (`long double`), затем выделяется память для массива, который сразу же инициализируется значениями из списка `l`. После чего создается переменная `n` типа `size_t` и происходит вызов C-функции.

Использование оберточных функций оправданно в большинстве случаев, так как позволяет спрятать рутинные действия по инициализации аргументов и указанию типов данных, предоставляя пользователю удобный интерфейс.

4. ГЕНЕРАЦИЯ ПСЕВДОСЛУЧАЙНЫХ ЧИСЕЛ

Получение истинно случайных чисел представляет достаточно трудную задачу. Обычно для этого используют разнообразные физические процессы. Основной проблемой генераторов истинно случайных чисел является низкая интенсивность генерации случайных чисел [15]. Поэтому для практических целей используют генераторы псевдослучайных чисел [16–20].

Пакет SymPy не реализует отдельных генераторов псевдослучайных чисел, так как вся необходимая функциональность присутствует в стандартном модуле `random` и в подмодуле `numpy.random` библиотеки NumPy.

Функции обоих модулей основаны на алгоритме под названием *вихрь Мерсенна* [21], который генерирует псевдослучайные равномерно распределенные последовательности беззнаковых целых чисел. Данный алгоритм позволяет получить качественную последовательность псевдослучайных чисел, однако отличается сравнительно низкой производительностью ввиду громоздкости самого алгоритма. В настоящее время появился ряд альтернативных алгоритмов [22–25], которые также генерируют качественные последовательности псевдослучайных чисел, но при этом выигрывают в быстродействии.

Современные алгоритмы генераторов псевдослучайных чисел используют побитовые логические операции и операции сдвига, поэтому естественным выбором для реализации таких алгоритмов являются системные языки программирования, обеспечивающие минимум абстракций в пользу максимального уровня быстродействия. Большинство разработчиков данных алгоритмов предоставляют также примеры реализаций на языках C или C++.

Такие реализации представляют собой компактные функции, сигнатура которых имеет следующий вид:

```
uint32_t generator(uint32_t seed[]);
/* или */
uint64_t generator(uint64_t seed[]);
```

где массив `seed` представляет собой начальные значения для инициализации генератора. Для генерации последовательности псевдослучайных чисел данную функцию достаточно вызвать в цикле N раз.

```
void rand_n(uint64_t N, uint64_t seed[],
↪ uint64_t res[]) {
  for (uint64_t i=0; i < N; ++i) {
      res[i] = generator(seed);
  }
}
```

Внутреннее состояние генератора определяется набором чисел `seed` и сохраняется от вызова к вызову, так как массив `seed` передается по ссылке.

Для получения чисел из полуинтервала [0, 1) достаточно нормировать генерируемые числа. Нижеследующий код показывает как это сделать.

```
double normed_gen(uint64_t seed[]) {
    return (double) generator(seed) /
    ↪ (double) UINT64_MAX;
}
```

4.1. Структура библиотеки

Авторами была реализована компактная библиотека на языке C [26], в которой был собран ряд современных генераторов псевдослучайных чисел [27, 22–25]. Библиотека имеет следующую структуру.

• В директории `src` располагаются файлы с исходным кодом, реализующие различные алгоритмы генераторов псевдослучайных чисел. Файлы названы именем алгоритма, реализация которого содержится внутри.

• В директории `include` содержится единственный файл заголовка, в котором объявлены все функции, реализованные в библиотеке.

• В директории `tools` находятся C-программа, реализующая утилиту командной строки `random`, с помощью которой можно запустить любой генератор для вывода сгенерированной последовательности на печать.

• Для сборки библиотеки и утилиты под операционной системой типа Unix написан makefile, а для сборки под ОС Windows командный bat-файл.

• Результатом компиляции и сборки будут файлы разделяемой и статической библиотек, расположенные в директориях `lib/shared` и `lib/static` соответственно. Также в директории `bin` будет собрана командная утилита `random`

4.2. Обертка библиотеки с помощью `ctypes`

Вышеописанная библиотека была интегрирована в среду Python/SymPy с помощью стандартного модуля `ctypes` и оформлена в виде Python-модуля под названием `crandom`. Все функциональные возможности модуля реализованы в файле `crandom.py` в виде класса `Random`.

Для корректного функционирования требуются стандартные модули `random`, `typing`, `ctypes`, `sys` и `os`. Также для генерации массивов псевдослучайных чисел требуется библиотека NumPy.

Рассмотрим основные возможности `crandom` на примерах. Для запуска примеров использовал-

ся дистрибутив языка Python 3.6.8 Miniconda и интерактивная оболочка Jupyter 4.4.0.

Работа с модулем начинается с выбора и инициализации генератора. Рассмотрим пример:

```
import crandom
gen = crandom.Random('xorshift+')
gen.set_seed([233, 43])
```

Здесь мы создали объект `gen` который будет использовать алгоритм *xorshift+* своей работе. Также мы инициализировали генератор, передав ему два целых числа с помощью функции-метода `set_seed`. Если генератор не инициализировать явным вызовом `set_seed`, то будет использована функция `randint` из стандартного модуля `random`. Также следует отметить, что при выборе начальных значений следует придерживаться ряда рекомендаций [28] и числа 233, 43 были выбраны только для того, чтобы не загромождать пример.

Состояние генератора сохраняется в атрибуте `seed` объекта `gen`. В зависимости от типа генератора `seed` может быть как единственным беззнаковым целым числом, так и последовательностью беззнаковых целых чисел. Вызванная без аргументов, функция-метод `set_seed` самостоятельно определяет сколько целых чисел необходимо для инициализации.

После того, как объект `gen` создан и инициализирован, его можно использовать для получения последовательности псевдослучайных чисел заданного размера. Сделать это можно следующими способами.

```
r = gen.generate(size=10)
r = gen.generate(size=10, type=float)
r = gen(size=10)
```

При первом вызове будет сгенерированна последовательность из 10 беззнаковых целых чисел. При втором вызове необязательному аргументу `type` передано `float`, что приводит к генерации последовательности чисел с плавающей запятой из полуинтервала [0, 1). Наконец третья строка показывает, что необязательно использовать функцию-метод `generate` так как в классе определен метод `__call__`, и сам объект можно вызывать как функцию.

Генерацию массива псевдослучайных чисел полностью осуществляет функция на языке C (для каждого генератора своя). Затем сгенерированный массив конвертируется в numpy-массив. Для конвертации вызывается функция `np.array` с опцией `copy=False`, что позволяет не копировать массив в памяти, а заместить по месту (in place).

Состояние генератора сохраняется средствами Python-программы. После того, как последовательность сгенерирована, в атрибут `seed` записываются последние элементы этой последовательности. Они будут использованы в качестве новых начальных значений.

Если приоритетом является экономия памяти, то генератор можно использовать в режиме итератора, так как в классе реализованы специальные функции-методы `_next_` и `_iter_` Следующий пример иллюстрирует как это сделать.

```
gen.set_iterator(10, int)
for i in gen:
    print(i)
```

Инициализация итератора осуществляется функцией `set_iterator`. В качестве аргументов указывается количество чисел, которое должен произвести генератор, и тип чисел (`int` или `float`). Затем объект `gen` можно использовать в цикле, как стандартный python-итератор. При этом работает цикл, реализованный на Python, в результате чего производительность ниже чем при использовании функции `generate`. Состояние генератора сохраняется также, как и при использовании `generate`.

4.3. Тестирование производительности

Использование C-функций позволяет достичь высокой производительности. Сравним, например, работу нашего генератора с генератором из библиотеки NumPy `randint`. Быстродействие измеряется командой `%timeit`, встроенной в интерактивные оболочки *iPython* и *Jupyter*. В качестве аргумента ей передается фрагмент кода, быстродействие которого следует замерить. В качестве результата распечатывается значение среднего времени работы кода, среднеквадратичное отклонение и количество выполнений кода.

Для получения последовательности 64-битных беззнаковых целых чисел, при вызове функции `randint` необходимо указать значение `np.uint64` аргумента `dtype`, а также указать нижнюю (`low`) и верхнюю (`high`) границы.

Для обеих функций команда %timeit произвела 7 запусков по 10000 повторений в каждом. Среднее время работы функции `randint` составило 85.4 со стандартным отклонением в 1.67 микросекунды. Для функции `generate` – 39.4 микросекунды, со стандартным отклонением 1.18 микросекунды.

Так как в библиотеке NumPy генераторы также реализованы на языке C, то полученную разницу можно объяснить большей эффективностью алгоритма *xorshift*. Следует также отметить,

что компиляция библиотеки выполнялась с ключом оптимизации `-O3`.

Отметим, что в стандартном модуле `random` нет функции, позволяющей сгенерировать последовательность целых чисел. Взамен этого можно воспользоваться многократным вызовом функции randint и списковой сборкой, что будет заведомо медленней NumPy-версии (замер времени дает значение 11.6 миллисекунды).

Для проверки корректности реализации генераторов был проведен ряд визуальных тестов. Были построены следующие диаграммы:

- диаграмма рассеяния (scatter plot);
- диаграмма лага (Lag-plot);
- диаграмма автокорреляции в зависимости от лага (auto-correlation function plot, ACF-plot).

Данные визуальные тесты позволяют оценить насколько полученная последовательность псевдослучайных чисел является независимо распределенной и выявить лишь грубые ошибки. В качестве более строгих тестов были использованы наборы тестов DieHarder [29], TestU01 [30, 31], PractRand [32] и gjrand [33]. Отчеты по тестам DieHarder доступны в репозитории [26].

5. ЗАКЛЮЧЕНИЕ

Созданная нами библиотека и модуль для ее интеграции в среду SymPy/Python могут быть легко расширены добавлением новых функций на языке C и соответствующих оберточных функций на языке Python.

Отметим также, что в случае генераторов псевдослучайных чисел выбор модуля `ctypes` был обоснован, так как реализуемые алгоритмы используют побитовые операции и примитивные типы данных, поэтому их реализация полностью на системном языке программирования дает существенное увеличение производительности и уменьшение расхода памяти.

БЛАГОДАРНОСТИ

Публикация подготовлена при поддержке Программы РУДН “5-100”.

СПИСОК ЛИТЕРАТУРЫ

1. *Lamy R.* Instant SymPy Starter. Packt Publishing, 2013. 52 p.
2. *Слаткин Б.* Секреты Python. М.: Вильямс. 2017. 272 с.
3. *Любанович Б.* Простой Python. Современный стиль программирования. Бестселлеры O’Reilly. М.: Питер, 2019. 480 с.
4. *Кулябов Д.С., Королькова А.В., Севастьянов Л.А.* Новые возможности второй версии пакета компьютерной алгебры Cadabra // Программирование. 2019. № 2. С. 41–48.
5. *Aladjev V., Bogdevicius M.* Maple: Programming, Physical and Engineering Problems. 2006. 403 p. ISBN: 1596820802.
6. *Corless R.M.* Essential Maple 7: An Introduction for Scientific Programmers. Springer Science and Business Media, 2007. 282 p. ISBN: 9780387215570.
7. *Idris I.* NumPy Cookbook. Packt Publishing, 2012. 226 p.
8. *Oliphant T.E.* Guide to NumPy. 2 edition. CreateSpace Independent Publishing Platform, 2015. 364 p.
9. *Oliphant T.E.* Python for Scientific Computing // Computing in Science and Engineering. 2007. V. 9. № 3. P. 10–20.
10. *Behnel S., Bradshaw R., Citro C.* et al. Cython: The Best of Both Worlds // Computing in Science and Engineering. 2011. mar. V. 13. № 2. P. 31–39.
11. *Smith K.* Cython. A Guide for Python Programmers. O’Reilly Media, 2015. 238 p.
12. *Lam S.K., Pitrou A., Seibert S.* Numba: a LLVM-based Python JIT compiler // Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. LLVM '15. Austin, Texas: ACM Press, 2015. nov. P. 7.1–6.
13. *Spreitzenbarth M., Uhrmann J.* Mastering Python Forensics. Packt Publishing, 2015. 192 p.
14. *.Клеменс Б.* Язык C в XXI веке. М.: ДМК Пресс, 2017. 376 с.
15. *Galton F.* Dice for statistical experiments // Nature. 1890. V. 42. № 1070. P. 13–14.
16. *Кнут Д.Э.* Искусство программирования. 3 изд. М.: Вильямс, 2001. Т. 2. 832 с.
17. *Дроздова И.И., Жилин В.В.* Генераторы случайных и псевдослучайных чисел // Технические науки в России и за рубежом: материалы VII Междунар. науч. конф. М.: Буки-Веди, 2017. nov. С. 13–15.
18. *Колчин В.Ф., Севастьянов Б.А., Чистяков В.П.* Случайные размещения. М.: Наука, 1976. 224 с.
19. *Тюрин Ю.Н., Макаров А.А.* Статистический анализ данных на компьютере / Под ред. В.Э. Фигурнова. М.: ИНФРА, 1998. 528 с.
20. *Gevorkyan M.N., Demidova A.V., Korolkova A.V.* et al. Pseudo-random number generator based on neural network // Selected Papers of the 8th International Conference “Distributed Computing and Gridtechnologies in Science and Education” / Ed. by Vladimir Korenkov, Andrey Nechaevskiy, Tatiana Zaikina, Elena Mazhitova. Vol. 2267 of CEUR Workshop Proceedings. Dubna, 2018. sep. P. 568–572.
21. *Matsumoto M., Nishimura T.* Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator // ACM Trans. Model. Comput. Simul. 1998. V. 8. № 1. P. 3–30.
22. *Marsaglia G.* Xorshift RNGs // Journal of Statistical Software. 2003. V. 8. № 1. P. 1–6.
23. *Panneton F., L’Ecuyer P.* On the Xorshift Random Number Generators // ACM Trans. Model. Comput. Simul. 2005. V. 15. № 4. P. 346–361.

24. *Boldi P., Vigna S.* On the Lattice of Antichains of Finite Intervals // Order. 2018. mar. V. 35. № 1. P. 57–81.
25. PCG: A Family of Simple Fast Space-EfficientStatistically Good Algorithms for Random Number Generation: Rep.: HMC-CS-2014-0905 / Harvey Mudd College; Executor: Melissa E O'Neill. Claremont, CA: 2014.
26. *Gevorkyan M.N., Kulyabov D.S., Korolkova A.V., Sevastianov L.A.* Random Number Generators for Computer Algebra Systems. 2019. URL: https://bitbucket.org/yamadharma/articles-2019-rng-generator-code.
27. *Rose G.G.* KISS: A bit too simple // Cryptography and Communications. 2018. V. 10. № 1. P. 123–137.
28. *Jones D.* Good practice in (pseudo) random number generation for bioinformatics applications. 2010.
29. *Brown R.G., Eddelbuettel D., Bauer D.* Dieharder: A Random Number Test Suite. 2017. URL: http://www.phy.duke.edu/~rgb/General/rand_rate.php.
30. *L'Ecuyer P., Simard R.* TestU01: A C library for empirical testing of random number generators // ACM Transactions on Mathematical Software (TOMS). 2007. V. 33. № 4. P. 22.
31. *L'Ecuyer P., Simard R.* TestU01 – Empirical Testing of Random Number Generators. 2009. URL: http://simul.iro.umontreal.ca/testu01/tu01.html.
32. *Doty-Humphrey C.* PractRand official site. 2018. URL: http://pracrand.sourceforge.net/.
33. Gjrand random numbers official site. 2014. URL: http://gjrand.sourceforge.net/.