Volume 4, Issue 10, October- 2026 **ISSN(E):** 2938-3773

Web of Discoveries: Journal of Analysis and Inventions
webofjournals.com/index.php/3

AUTOMATING THE SOLUTION OF SYSTEMS OF LINEAR EQUATIONS AND ASSESSING COMPUTATIONAL EFFICIENCY

Kamalova Nilufar Ilhomovna
Associate Professor, Department of "Applied Mathematics and Programming Technologies", Bukhara State University; Doctor of Philosophy (PhD) in Pedagogical Sciences

Azimova Gulasal Shavqiddin qizi
Student, Department of "Applied Mathematics and Programming Technologies", Bukhara State University

Abstract
The article considers the problems of automating the process of solving systems of linear equations and assessing computational efficiency. Based on the Python programming language and the Tkinter and NumPy libraries, a software tool has been developed that allows the user to enter matrix coefficients through a graphical interface, solve the system, and view the execution time in milliseconds. How the computation time and the accuracy of the solution change as the system size n increases was analyzed experimentally on random systems from n=3 to n=2000; the time was measured using the time. perf_counter() function. The program can be used in the educational process as a tool for checking examples that students have solved independently.

Keywords: System of linear equations, matrix, automation, computational efficiency, Python, Tkinter, NumPy, LU decomposition, numerical methods.

Introduction

Solving systems of linear equations is one of the central problems of computational mathematics and is widely used in physics, economics, engineering, chemistry, and data analysis. Many problems, such as the discretization of differential equations, the least squares method, and the analysis of electrical circuits and networks, ultimately reduce to solving a system of the form $Ax = b$.

The main method for solving such systems directly is Gaussian elimination and the equivalent LU decomposition; their theoretical foundations are set out in the textbooks by Golub and Van Loan [2] and by Trefethen and Bau [3]. In his monograph, Higham [1] analyzed the numerical stability of Gaussian elimination, that is, the effect of rounding errors on the result, through the concepts of backward error and the growth factor. That work substantiates the need to take into account the numerical stability of an algorithm, along with its theoretical correctness, in practical computations.

For large sparse systems, exploiting the structure of the matrix is of great importance. George and Liu [4] studied how, when solving sparse positive definite systems by direct methods, to take into account the positions of the nonzero elements and to reduce the appearance of new nonzero elements (fill-in) during the computation. In practice, such algorithms are used through

Licensed under a Creative Commons Attribution 4.0 International License.

Web of Discoveries: Journal of Analysis and Inventions
webofjournals.com/index.php/3

ready-made libraries – LAPACK [5] and NumPy [6], which relies on it: the numpy. linalg.solve() function solves a dense square system using LAPACK's LU decomposition routine.

In the educational process, students first learn to solve a system by hand, but as the number of unknowns grows, hand computation becomes time-consuming and error-prone. Therefore, there is a need for a simple tool that allows results to be checked quickly and shows how computation time depends on the size of the system.

The aim of this article is to develop a software tool based on Python, Tkinter [7], and NumPy that automatically solves systems of linear equations, and to assess experimentally how the computation time and the accuracy of the solution change as the system size *n* increases.

Problem Statement

Solving systems of linear equations by hand becomes more complicated as the number of equations and unknowns grows, takes a great deal of time, and increases the likelihood of computational errors. Therefore, automating the process with a computer and assessing its efficiency is a relevant problem.

A system of *n* linear equations with *n* unknowns is written in matrix form as follows:

$$Ax = b \qquad (1)$$

Here *A* is the coefficient matrix, *x* is the vector of unknowns, and *b* is the vector of constant terms.

The following tasks are set in this work:

1) to create a graphical interface that, once the user enters the number of rows (*M*) and columns (*N*), dynamically generates cells for the matrix *A* and the vector *b* of the corresponding size;
2) to validate the entered data (the size, whether the matrix is square, whether the cells contain numbers, whether the determinant is zero) and to warn the user with an appropriate message;
3) to solve the system using NumPy and display the execution time in milliseconds;
4) to assess experimentally how the computation time and the accuracy of the solution change as the system size increases.

Mathematical Model

In the general case, the system consists of *m* equations and *n* unknowns, and its expanded algebraic form is as follows:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \qquad (2)$$

In matrix form, this system is written as follows:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \qquad (3)$$

Licensed under a Creative Commons Attribution 4.0 International License.

Web of Discoveries: Journal of Analysis and Inventions
webofjournals.com/index.php/3

Here a_{ij} are the matrix elements, i is the equation (row) number, j is the unknown (column) number; A is the $m \times n$ coefficient matrix, x is the $n \times 1$ column of unknowns, and b is the $m \times 1$ column of constant terms.

For a unique solution to exist, it is necessary and sufficient that $m = n$ (a square matrix) and $\det A \neq 0$. This is exactly the case considered in the program: if $M \neq N$ or the matrix is singular, the user is notified of the error.

In practice, the solution is found not by the formula $x = A^{-1}b$ but by an algorithm based on LU decomposition. NumPy's linalg.solve() function (LAPACK) performs LU decomposition with row interchanges (partial pivoting) and then solves two triangular systems in succession:

$$PA = LU, \qquad Ly = Pb, \qquad Ux = y \tag{4}$$

where P is the permutation matrix, L is the lower triangular matrix, and U is the upper triangular matrix. The number of arithmetic operations of the algorithm is approximately

$$W(n) \approx \frac{2}{3}n^3 \tag{5}$$

as shown in [1, 2]; therefore, for sufficiently large n, the computation time is expected to grow in proportion to n^3.

To assess the accuracy of the solution, systems with a known solution were used: taking $x^* = (1, 1, \ldots, 1)^T$, the vector $b = Ax^*$ was computed, and the relative error was determined for the computed solution $\hat{x}$:

$$\delta = \frac{\|\hat{x} - x^*\|_2}{\|x^*\|_2} \tag{6}$$

Conceptual Model of the Program

The structure of the software tool is shown in Figure 1. The model consists of five interrelated parts, whose functions are described in turn below.

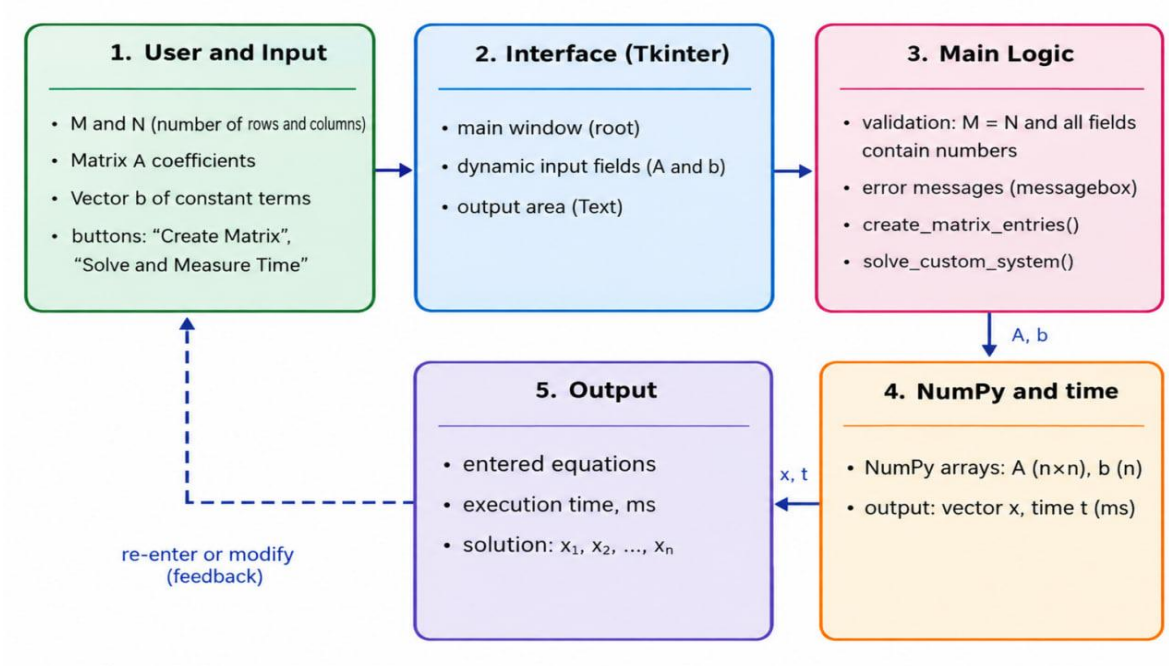


Figure 1. Conceptual model of automating the solution of systems of linear equations and assessing computational efficiency

Licensed under a Creative Commons Attribution 4.0 International License.

Web of Discoveries: Journal of Analysis and Inventions
webofjournals.com/index.php/3

1. User and input data. The user enters the number of rows (M) and columns (N), the coefficients of the matrix A, and the vector b of constant terms, and then clicks the "Create matrix" and "Solve and measure time" buttons. As a result, the initial data and commands needed for the computation are formed.

2. User interface (Tkinter). Using Python's Tkinter library, the main window, the control elements (Entry, Button, Label), and the dynamic cells for A and b are created. The user conveniently enters the data into the cells on the screen.

3. Core logic. This part consists of two functions: create_matrix_entries() checks the dimensions, displays the cells on the screen, and prepares the variables; solve_custom_system() checks that the matrix is square, reads the entered values, finds the solution with NumPy, measures the time, and formats the result.

4. NumPy and time. The entered numbers are converted to numpy.array (A is an $n \times n$ matrix, b is a vector of length n); numpy.linalg.solve() solves the system, and time.perf_counter() [8] measures the time spent on the computation. As a result, the solution vector x and the timing figure are obtained.

5. Result (output). The text field (Text) displays the entered equations, the execution time (ms), and the found solutions $x_1, x_2, \ldots, x_n$. If necessary, the user can re-enter or modify the data through feedback.

6. Program Code and Its Explanation

The main functions of the program are shown in Figure 2. The parts that create the interface elements (the window, buttons, and text field) and format the result consist of similar simple calls and are therefore not shown in the article.

```
import time
import tkinter as tk
from tkinter import messagebox
import numpy as np
def create_matrix_entries():
    global entries_A, entries_b
    for w in frame_matrix.winfo_children():
        w.destroy()  # clear the old cells
    try:
        M, N = int(entry_rows.get()), int(entry_cols.get())
```


Licensed under a Creative Commons Attribution 4.0 International License.

```
        if M < 2 or N < 2:
            raise ValueError
        if M > 8 or N > 8:
            messagebox.showwarning("Warning", "Do not exceed a size of 8x8!")
        draw_labels(M, N)

        entries_A = [[make_entry(i + 1, j) for j in range(N)] for i in range(M)]
        entries_b = [make_entry(i + 1, N + 1) for i in range(M)]
        btn_solve.pack(pady=10)
    except ValueError:
        messagebox.showerror("Error", "M and N must not be less than 2!")

def solve_custom_system():
    try:
        M, N = int(entry_rows.get()), int(entry_cols.get())
        if M != N:
            messagebox.showerror("Error", "The matrix must be square (M = N)!")
            return
        A = np.array([[float(e.get()) for e in row] for row in entries_A])
        b = np.array([float(e.get()) for e in entries_b])
        t0 = time.perf_counter()
        x = np.linalg.solve(A, b)  # solve and measure the time
        ms = (time.perf_counter() - t0) * 1000
        show_result(A, b, x, ms)
    except np.linalg.LinAlgError:
        messagebox.showerror("Error", "The matrix is singular (det = 0)!")
    except ValueError:
        messagebox.showerror("Error", "Enter a number in every cell!")
```

Figure 2. Main functions of the program: create_matrix_entries() and solve_custom_system()

- **Importing the libraries.** tkinter is used to create the graphical window, messagebox to display error and warning messages to the user, numpy for matrix computations, and time to determine the execution time. Tkinter is the standard interface toolkit included in Python [7].
- **Creating the main window.** The main window of the program is created with root = tk.Tk(). The window size is 600×650 pixels and its title is set to "Solving a System of Equations by Manual Input"; the general font type is set once using option_add().
- **Creating the cells (create_matrix_entries).** The function dynamically creates the matrix cells based on the values of *M* and *N* entered by the user. If the size is smaller than 2×2, an error message is shown; if it is larger than 8×8, a warning is displayed that the cells may not fit on the screen.
- **Matrix and constant-term cells.** tk.Entry cells for entering numbers are arranged in a grid at the intersection of each row and column; at the end of each row, an equals sign (=) and a separate cell for the constant term (the vector *b*) are created. The initial value is set to 0.

Web of Discoveries: Journal of Analysis and Inventions
webofjournals.com/index.php/3

Licensed under a Creative Commons Attribution 4.0 International License.

Web of Discoveries: Journal of Analysis and Inventions
webofjournals.com/index.php/3

- **Solving and measuring time (solve_custom_system).** The function first checks the condition $M = N$, then converts the values in the cells to the float type and builds the NumPy arrays A and b. Using time.perf_counter(), only the start and end times of the np.linalg.solve(A, b) call are recorded, and the difference is converted to milliseconds.
- **Error handling.** If a non-numeric value is entered in a cell, ValueError is caught, and if the matrix is exactly singular, np.linalg.LinAlgError is caught; in either case the user is shown the corresponding message.
- **Displaying the result.** The text form of the entered equations, the execution time, and the solutions x_1, x_2, … are displayed in the txt_result text field (in the Consolas font, to four decimal places).

Computational Experiment and Results

Because the interface limits the number of cells to 8×8, the efficiency experiment was conducted in a separate program, benchmark.py (Figure 4). For each $n \in \{3, 5, 8, 10, 20, 50, 100, 200, 500, 1000, 2000\}$, a random matrix A with elements drawn from the standard normal distribution (NumPy generator, seed = 42), $x^* = (1, \ldots, 1)^T$, and $b = Ax^*$ were taken. For each size, a warm-up call was performed first, then the solve() function was repeated from 5 to 2000 times depending on n, the time was measured with time.perf_counter(), and the median was computed.

```
import json
import platform
import statistics
import time
import numpy as np
SIZES = [3, 5, 8, 10, 20, 50, 100, 200, 500, 1000, 2000]
rng = np.random.default_rng(42)  # for reproducible results
def repeats(n):  # more repetitions for small n
    if n <= 20:
        return 2000
    if n <= 100:
        return 500
    if n <= 500:
        return 50
    return 10 if n <= 1000 else 5
# First ("cold") call: this is exactly what the user sees in the GUI program
```


Licensed under a Creative Commons Attribution 4.0 International License.

```
A0 = rng.standard_normal((3, 3))
b0 = A0 @ np.ones(3)
t0 = time.perf_counter()
np.linalg.solve(A0, b0)
cold_ms = (time.perf_counter() - t0) * 1000

rows = []
for n in SIZES:
    A = rng.standard_normal((n, n))
    x_true = np.ones(n)
    b = A @ x_true
    np.linalg.solve(A, b)  # warm-up
    times = []
    for _ in range(repeats(n)):
        t = time.perf_counter()
        x = np.linalg.solve(A, b)
        times.append((time.perf_counter() - t) * 1000)
    err = np.linalg.norm(x - x_true) / np.linalg.norm(x_true)
    rows.append({"n": n, "median_ms": statistics.median(times),
                 "rel_error": float(err)})

print(f"Python {platform.python_version()}, NumPy {np.__version__}, "
      f"{platform.system()} {platform.release()}")
print(f"First call (3x3): {cold_ms:.4f} ms")
print(f"{'n':>6} {'median, ms':>14} {'relative error':>16}")
for r in rows:
    print(f"{r['n']:>6} {r['median_ms']:>14.4f} {r['rel_error']:>16.2e}")

with open("benchmark_results.json", "w") as f:
    json.dump({"cold_ms": cold_ms, "rows": rows}, f, indent=1)
```

Figure 4. Program for measuring efficiency (benchmark.py)

Web of Discoveries: Journal of Analysis and Inventions
webofjournals.com/index.php/3

Licensed under a Creative Commons Attribution 4.0 International License.

Web of Discoveries: Journal of Analysis and Inventions
webofjournals.com/index.php/3

The results are presented in Table 1 and Figure 5.

Table 1. Time and accuracy of solving systems of linear equations of various sizes

System size (n×n)	Execution time (median), ms	Relative error δ
3×3	0.0059	$1.9 \cdot 10^{-16}$
5×5	0.0061	$9.9 \cdot 10^{-17}$
8×8	0.0078	$6.3 \cdot 10^{-16}$
10×10	0.0086	$2.6 \cdot 10^{-15}$
20×20	0.0143	$9.4 \cdot 10^{-16}$
50×50	0.0377	$6.0 \cdot 10^{-15}$
100×100	0.1282	$3.4 \cdot 10^{-14}$
200×200	0.6075	$1.9 \cdot 10^{-13}$
500×500	5.340	$1.1 \cdot 10^{-13}$
1000×1000	31.37	$1.4 \cdot 10^{-13}$
2000×2000	196.3	$8.1 \cdot 10^{-12}$

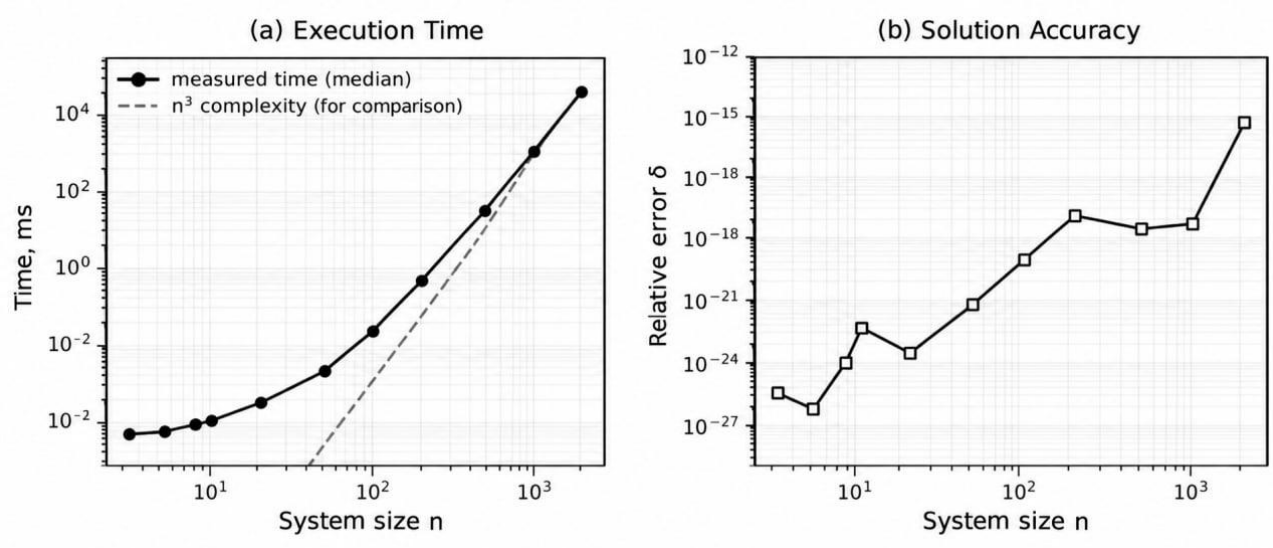


Figure 5. Execution time (a) and relative error (b) as a function of system size (logarithmic scale)

As Table 1 shows, for small sizes ($n \leq 10$) the time hardly changes (about 0.0059–0.0086 ms): in this case the overhead of calling the function and preparing the arrays dominates over the computation itself. For comparison, the first call in Figure 3 took 1.35 ms. For $n \geq 200$ the time grows rapidly: when n doubled from 1000 to 2000, the time increased from 31.37 ms to 196.3 ms, that is, by a factor of about 6.3 (a factor of 8 would be expected under the n^3 law). For $n \geq 200$, a linear approximation on a log–log scale gave the exponent $T \sim n^{2.5}$. The deviation from the theoretical n^3 is probably due to the LAPACK/BLAS libraries using memory more efficiently on large matrices.

Licensed under a Creative Commons Attribution 4.0 International License.

Web of Discoveries: Journal of Analysis and Inventions
webofjournals.com/index.php/3

Regarding the accuracy of the solution: the relative error did not exceed $8.1 \cdot 10^{-12}$ for any size. At $n = 3$ it was on the order of machine epsilon ($\approx 2.2 \cdot 10^{-16}$) and grew slowly as n increased (Figure 5, b); this can be explained by the accumulation of rounding errors and the growth of the condition number of random matrices. It should be noted that the program window displays the result to four decimal places (.4f format) – this is only the display precision and does not reflect the computational accuracy.

Use of the Program in the Educational Process

The most convenient way to use the program in the educational process is for the student to first solve the system independently and then check the result by entering the coefficients into the program. For example, systems of size 3×3 or 4×4 can be entered and the answers compared. It is also a useful experiment to increase the size gradually and observe how the computation time changes. This ensures that the student does not simply obtain a ready-made answer but understands the process by seeing it in action.

Discussion

The main advantage of the program is that it brings theory and practice together: as soon as the user enters the coefficients, NumPy solves the system, and the timer shows the time spent in milliseconds. At the same time, the program has the following limitations:

• only square ($M = N$) systems are solved; for rectangular systems, the least squares method (numpy.linalg.lstsq) could be added;

• so that the cells fit on the screen, a size of up to 8×8 is recommended in the interface, while larger systems were tested in separate programs such as benchmark.py;

• LinAlgError arises only for an exactly singular matrix; for nearly singular (ill-conditioned) systems, the program may return a result with a large error without raising an error. To prevent this, it is advisable to compute the condition number with numpy.linalg.cond and warn the user;

• only the direct method (LU decomposition) was used, with no comparison against iterative methods; the measurements were carried out on a single computer.

Conclusion

In this study, a Python–Tkinter–NumPy software tool was developed for automating the process of solving systems of linear equations and assessing computational efficiency. The tool allows the user to enter matrix coefficients and constant terms through a graphical interface, to solve the system quickly and accurately with the LU-decomposition-based numpy.linalg.solve() function, and to view the execution time in milliseconds.

According to the experimental results, for small systems ($n \leq 10$) the computation time is almost independent of the system size, whereas for $n \geq 200$ it grows rapidly, exhibiting a law close to $n^{2.5}$; the relative error was below $8.1 \cdot 10^{-12}$ for all sizes. The time of the first call was found to be considerably higher than the median of repeated measurements; therefore, repeated measurements are necessary when assessing efficiency.

The software tool can be used in teaching higher mathematics and programming as a didactic aid that helps link theoretical knowledge with practice, check errors in manual computation,

Licensed under a Creative Commons Attribution 4.0 International License.

and analyze computational efficiency. In the future, it is planned to add to the program iterative methods (Jacobi, Seidel), a condition number check, rectangular systems, output of timing results in graphical form, and saving of the entered data.

References

1. Higham N. J. Accuracy and Stability of Numerical Algorithms. 2nd ed. – Philadelphia: SIAM, 2002.
1. Golub G. H., Van Loan C. F. Matrix Computations. 4th ed. – Baltimore: Johns Hopkins University Press, 2013.
2. Trefethen L. N., Bau D. Numerical Linear Algebra. –Philadelphia: SIAM, 1997.
3. George A., Liu J. W. H. Computer Solution of Large Sparse Positive Definite Systems. – Englewood Cliffs: Prentice-Hall, 1981.
4. Anderson E., Bai Z., Bischof C. et al. LAPACK Users' Guide. 3rd ed. – Philadelphia: SIAM, 1999.
5. Harris C. R., Millman K. J., van der Walt S. J. et al. Array programming with NumPy // Nature. – 2020. –Vol. 585. – P. 357–362.
6. Python Software Foundation. tkinter – Python interface to Tcl/Tk. Python 3 Documentation. – URL: https://docs.python.org/3/library/tkinter.html
7. Python Software Foundation. time – Time access and conversions (time.perf_counter). Python 3 Documentation. – URL: https://docs.python.org/3/library/time.html

Web of Discoveries: Journal of Analysis and Inventions
webofjournals.com/index.php/3

Licensed under a Creative Commons Attribution 4.0 International License.