Abstract—The paper deals with the classical fiber bundle model of equal load sharing, sometimes referred to as the Daniels’ bundle or the democratic bundle. Daniels formulated a multidimensional integral and also a recursive formula for evaluation of the strength cumulative distribution function. This paper describes three algorithms for evaluation of the recursive formula and also their implementations with source codes in the Python high-level programming language. A comparison of the algorithms is provided with respect to execution time. Analysis of orders of magnitudes of addends in the recursion is also provided.

Keywords—Daniels bundle model, equal load sharing, Python, mpmath.

I. INTRODUCTION

This paper deals with the classical fiber bundle model with equal load sharing, sometimes referred to as the Daniels bundle model [1] or the democratic bundle [4]. This model is significant for the strength of fibrous materials and composites, and the generally random strength of quasi-brittle structures. The model is also relevant for the analysis of the reliability of various parallel systems (computer components, infrastructure etc.). Daniels formulated a multidimensional integral and also recursive formula for the evaluation of the strength distribution function. In the same paper, he showed that the distribution of the strength of the bundle, \( G_n(x) \), tends to Gaussian distribution under quite broad conditions and he gave closed formulas for the mean value and standard deviation of the Gaussian distribution. Sornette [4] later confirmed this result using a Kolmogorov theorem.

The convergence of a random strength to Gaussian distribution is very slow in terms of the number of fibers and therefore, Smith [3] proposed a corrected term for the mean value that improves the original Daniels formula for small bundles. Even though the knowledge of the asymptotic form of \( G_n \) for the number of fibers \( n \to \infty \) is important, the normality does not hold in the tails of the distribution and it also does not hold when there is a small number of parallel components in the system (fibers). The real cumulative distribution function (CDF) \( G_n \) strongly deviates from the normal distribution for values of \( x \) far from the mean strength. Only a little is known about the behavior of the tails. The left tail \( (x \to 0) \) is of great importance for reliability considerations, however.

In the present paper, the authors describe an analysis of the recursive formula by Daniels. The advantage of the recursive formula is that it provides exact values of the CDF \( G_n \) for arbitrary values of \( x \) and for any number of fibers \( n \). The disadvantage is that a naïve implementation of the function makes it usable only for bundles as small as approx 20 fibers. However, we show that the number of arithmetic operations can be significantly decreased by exploiting the fact that some terms in the naïve recursive formula are repeated. The authors also describe a computer implementation carried out in the Python high-level programming language [5] using the NumPy [7] (scientific computing with arrays) and the mpmath [6] (library for real and complex floating-point arithmetic with arbitrary precision) packages. This implementation enables the calculation of cumulative distribution function (CDF) values for large numbers (thousands) of fibers in a bundle, including values deep in the left tail of the distribution (probabilities \( 10^{-600} \)). This computer program is used to accurately calculate the distribution functions \( G_n \) for bundles with Weibull fibers with the scale parameter \( s = 1 \), the varying number of fibers \( n \) and the varying shape parameter \( m \). The obtained results are stored in a new created database and compared to the available formulas [1]–[3]. The main motivation of the work is to formulate an improved analytical formula for the distribution function \( G_n \) that will be valid deep to the left tail where the real distribution strongly deviates from the Gaussian approximation.

II. BUNDLE STRENGTH

The cumulative distribution function, \( G_n \), of bundle strength formulated by Daniels [1] reads:

\[
G_n(x) = \sum_{k=1}^{n} (-1)^{k+1} \binom{n}{k} [F(x)]^k G_{n-k} \left( \frac{n x}{n - k} \right)
\]

where \( x \) is a value of random strength per fiber, \( X \), for which the probability \( P(X \leq x) = G_n(x) \) is evaluated, \( G_1(x) = F(x) \), \( G_0(x) \equiv 1 \), \( \binom{n}{k} \) is the standard Binomial coefficient and \( n \) is the number of fibers. The most frequently selected distribution function for a single fiber, \( F(x) \), is Weibull distribution. With no loss of generality, let us assume the bundles contain Weibullian fibers (elements in parallel) so that the CDF of the strength of single fiber is

\[
F(x) = 1 - e^{-(x/s)^m}
\]

where \( s \) is the scale parameter and \( m \) is the shape parameter.

III. IMPLEMENTATIONS OF THE RECURSIVE FORMULA

The implementations were carried out in the Python high-level programming language. Three approaches to
evaluate the recursive formula (1) will be described in the following text:
A. recursive implementation,
B. recursion with memoization, and
C. loop-based implementation (no recursion).

A. Recursive implementation

The recursive definition in (1) can be translated directly into Python as follows:

```python
import math
from scipy.misc import comb

def g(x, scale, shape, n):
  cdf = 1 - math.exp(-(x/scale)**shape)
  if n < 1:
    return 1.
  if n == 1:
    return cdf
  cdf_k = 1.
  res = 0.
  for k in range(1, int(n+1)):
    if k % 2 == 0:
      k nabla = k - k / 2
    else:
      k=lambda

  res += res_k
  return cdf_k

# set number of decimal places for multiprecision numbers
mp.mp.dps = 1000

# pre-conversion of frequently used values
MPF_ZERO = mp.mpf('0')
MPF_ONE = mp.mpf('1')
MPF_TWO = mp.mpf('2')
MPF_THREE = mp.mpf('3')

MPF_ZERO, MPF_ONE, ... are pre-converted values because repeated type conversions from floats, strings and integers are expensive. The precision used to evaluate \( G_n(x) \) up to \( n = 1500 \) was set to 1000 decimal places (3325 bits). This value was found to be sufficient while considering demands on the execution time and to requested accuracy. Any tests of optimal precision were not performed yet.

The Binomial coefficients \( \binom{n}{k} \) are calculated including the sign \((-1)^{k+1}\) and stored in the lower triangle of 2D array `binom_tab`.

```python
def get_binom_tab(n):
  binom_tab = np.zeros((n, n), dtype=object)
  for i in range(1, n+1):
    for j in range(1, 1+i):
      binom_tab[i-1, j-1] = (mp.binomial(i, j) * (-1)**(j+1))
  return binom_tab

This array can be precalculated for greater \( n \) and stored in a file on hard-disk. It is useful in case of repeated calculations.

The CDF of Weibull distribution is implemented consistently with \( F(x) \) in (2):

```python
def weib_cdf(x, shape, scale):
  ... Cumulative distribution function of Weibull distribution with two parameters (shape and scale).
  return MPF_ONE - mp.exp(-(x / scale)**shape)

The following code shows the implementation of \( G_n(x) \) with memoization:

```python
def gn_mp(x, scale, shape, n, binom_tab):
  ... Return value of CDF of a bundle strength considering Weibullian fibers.
  Parameters
  x : mpf
  Bundle strength
  scale : mpf
  Scale parameter of Weibull distribution
  shape : mpf
  Shape parameter of Weibull distribution
  n : mpf
  Number of fibers
  Returns
  out : mpf
  CDF value for the strength \( x \)
  Notes
```

```
Parameters can be float but it can cause inaccuracies in results.

Examples
---
>>> mp.mp.prettty = True
>>> mp.mp.dps = 30
>>> shape = mp.mpf(‘1.’)
>>> n, k = 980, 980
>>> n! = mp.exp(mp.mpf(‘-’))
>>> binom_tab = get binom tab(n fil)
>>> gn_mpx(x, scale, shape, n fil, binom tab)
0.00000035298591305074574682994

gn_arr = np.zeros (n, dtype=object)
cdf_arr = np.zeros (n, dtype=object)
x_arr = np.zeros(n, dtype=object)

for i in range(1, int(n)):
    x_arr[i-1] = mp.fraction(n, n - i) * x

def recursion_gn_mpx(x, val, scale, shape, n):
    index_n = int(n) - 1
    res = MPF.ZERO
    cdf = cdf_arr[index_n]
    if cdf == None:
        cdf = webcdf(x, val, shape, scale)
        cdf_arr[index_n] = cdf
    for k in range(1, int(n) + 1):
        gn = gn_arr[index_n, k - 1]
        if gn == None:
            cdf_k = cdf ** k
            comb = binom_tab[index_n, k - 1]
            if k == n:
                gn = (comb * cdf_k) *
                    recursion_gn_mpx(x, val, scale, shape, n)
            else:
                gn = comb * cdf_k # == G(0)(x) (= 1.0)
        gn_arr[index_n, k - 1] = gn
    res += gn
    return res


Arrays x_arr and cdf_arr contain precalculated values of x and F(x).

For each regularized measure of F(x), the implementation will raise error “RuntimeError: maximum recursion depth exceeded in cmp”. This error is caused by the default Python setting for maximum recursion depth – to increase it use import sys; sys.setrecursionlimit(value). The default value can be obtained by executing sys.getrecursionlimit(). The disadvantage of this implementation is that the memoized values are accessed repeatedly \((\frac{n-2}{n-1} - \frac{(n-2)(n-3)(n-4)}{4})\) times.

C. Loop-based implementation

The most efficient formulation of the algorithm, which is also a computer implementation among the three seems to be the following one. Let us divide \(1\) into three parts

\[
G_n(x) = \sum_{k=1}^{n} (-1)^{k+1} \binom{n}{k} \frac{F(x)}{S_k} G_{n-k} \left(\frac{x}{n-k} \right).
\]

At the highest level of recursion, formula (3) represents a summation over \(k = 1, \ldots, n\). Each of these addends calls for

a recursion – an evaluation of the recursion function \(G_n(x)\) with a new parameter \(n\) named here as \(n_k = n - k\).

By analyzing all arguments of random strength, \(x\), one can see that there are only \(n\) different values for which the distribution function of strength is evaluated, namely \(x_i = x \frac{n}{n-i}\), \(i = 1, \ldots, n\). The values of \(x\) are stored in an array named x_arr.

For each element of vector \(x\), one must compute the distribution function of the strength of one fiber, \(F(x)\). Let us now define a vector, \(F\), that contains the values of the basic CDF evaluated at points \(x_i\):

\[
F: \quad F_i = F(x_i) = F\left(\frac{x_i}{n-i}\right), \quad i = 1, \ldots, n
\]

This vector is precalculated and cached in computer memory as an array named cdf_arr at the beginning of computation. In later stages of computation, the elements of this vector are raised to integer powers \(k = 1, \ldots, n\).

The next ingredient is a lower triangular matrix \(B\), with \(n\) rows and \(n\) columns, pre-filled with binomial coefficients multiplied by alternating sign. Each element of the triangular matrix initially reads

\[
B: \quad B_{i,k} = (-1)^{k+1} \binom{n}{k}, \quad i = 1, \ldots, n \text{ and } \quad k = 1, \ldots, i
\]

This matrix is stored in an array named gn_arr.

Once these two ingredients are calculated, the algorithm continues with the following two loops (the algorithm uses in-place operations and updates values of \(B\) matrix):

1. Loop over \(n\) columns of the \(B\) matrix – starting with the first column \(k = 1\), each column \(k = 1, \ldots, n\) is multiplied by the \(k\)th power of elements \(F_i: B_{i,k} = B_{i,k} F_k^i\) for \(k = 1, \ldots, n\) and \(i = 1, \ldots, n\).

2. Loop over \(n - 1\) rows of the \(B\) matrix – starting with the row \(i = 2\), run a cycle over rows \(i = 2, \ldots, n\) that \(i\) sums the elements in the preceding row: \(S_{i-1} = \sum_{k=1}^{i-1} B_{i-1,k}\) and \((ii)\) use this value to update (multiply) this value with all elements of sub-diagonal number \(i\) within a cycle over columns \(j = 1, \ldots, n-i+1\): \(B_{i+j-1,j} = S_{i-1} B_{i+j-1,j}\).

After these two cycles are finished, the sum of \(n\) elements in the last row is the desired value of \(G_n(x)\):

\[
G_n(x) = S_n = \sum_{k=1}^{n} B_{n,k}
\]

Fig. 1 shows a diagram of the triangular array for the number of fibers \(n = 4\); factors featured in \(3\) are highlighted.

We note that the sums \(S_i\) of any row correspond to the strength distribution function of a fiber bundle with \(i\) fibers evaluated at load \(x \frac{n}{n-i}\):

\[
S_i = G_i(x \frac{n}{n-i}) = \sum_{k=1}^{i} B_{i,k} \quad i = 1, \ldots, n.
\]
The analysis of the recursive formula exploited in algorithm C seems to require the minimal possible number of arithmetic operations. The only option how to reduce the computing time could be to ignore elements of loop implementation that are insignificant for the resulting $G_n(x)$. To identify such elements, the analysis of orders $\log_{10} G_i$ of magnitudes of $G_i$ was performed. The following results were obtained for the number of fibers $n = 100$ and for the shape parameter $m = 6$. The mean value of strength of one fiber $\mu_x = 0.928$ (standard deviation $\sigma_x = 0.180$).

Fig. 2 shows results of $G_n$ for the various strengths $x$:
• $x = 0.01$ - the CDF value is $G_{100} = 2.384334 \times 10^{-847}$
• $x = 0.3$ - the CDF value is $G_{100} = 2.305565 \times 10^{-64}$
• $x = \mu_x = 0.98$ - the CDF value is very close to 1 and therefore we present the value of complement (survival probability) $1 - G_{100} = 2 \times 10^{-39}$
• $x = 1.5$ - the complement to the CDF value is $G_{100} = 2 \times 10^{-495}$.

In the right plot of Fig. 3, all values of $F(x)$ are close to 1 and this implies that the orders of magnitudes in the left triangular plot are, in fact, just the orders of the Binomial coefficients $B_{i,k} F^k S_i \approx B_{i,k} \cdot 1^k \cdot 1 = B_{i,k}$.

Based on the analysis of contributions to the value of $G_n$ from the $B$ matrix, we conclude that no systematic way of ignoring any elements of the $B$ matrix to further reduce the execution time can be suggested.

VI. CONCLUSION

The paper describes three approaches to the implementation of the recursive formula for evaluation of the cumulative distribution function of random bundle strength with: A) naïve Python recursive implementation, B) recursive implementation with memoization and mpmath, and C) loop-based implementation. The last implementation (C) significantly reduces the execution times and enables to calculate $G_n(x)$ for thousands of fibers and small probabilities.

The presented algorithms are used to create a database of CDF of bundle strengths for various $m$ and $n$ parameters. The database will serve as a data support for newly formulated analytical approximate formulas for the CDF of Daniels bundle strength.

ACKNOWLEDGMENT

The work was supported by projects FAST-S-14-2485 and by the project CZ.1.07/2.3.00/30.0005 of Brno University of Technology. The work of the second author has been supported by Czech Science Foundation under project number GACR GC13-19416J.

REFERENCES