International Journal of Mechanical, Industrial and Aerospace Sciences
ISSN: 2517-9950
Vol:9, No:6, 2015

Implementation of the Recursive Formula for Evaluation of the Strength of Daniels' Model

Václav Sadílek, Miroslav Vořechovský

***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 are 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.

V. Sadílek and M. Vořechovský is with the Institute of Structural Mechanics of Brno University of Technology, Czech Republic, e-mail: sadilek.v@fce.vutbr.cz, vorechovsky.m@fce.vutbr.cz.

II. Bundle Strength

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

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

where x is a value of random strength per fiber, X, for which the probability $P(X \leq x) = G_n(x)$ is evaluated, $G_1(x) \equiv F(x)$, $G_0(x) \equiv 1$, $\binom{n}{k}$ 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

$$F(x) = 1 - e^{-(x/s)^m} \quad (2)$$

where s is the scale parameter and m is the shape parameter.

III. Implemetations 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:

```
import math
from scipy.misc import comb

def Gn(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)):
        cdf_k *= cdf
        komb = comb(n, k) #binomial coefficient
        if k % 2 == 0.:
            komb = -komb
        res_1 = komb * cdf_k * Gn((n / (n - k)) * x,
                                  scale, shape, n - k)
        res += res_1

    cdf_k *= cdf
    if n % 2 == 0.:
        res -= cdf_k
    else:
        res += cdf_k

    return res
```

This Python implementation is limited to $n \approx 20$ due to the execution time and precision of double-precision floating-point format of numbers. The mpmath (library for real and complex floating-point arithmetic with arbitrary precision) package was used to remedy the precision problem however the problem with an explosion of the number of recursive calls is still present. This naïve implementation of the recursive formula (1) for G_n requires $2^n - 1$ function calls. For example, a bundle with $n = 50$ fibers necessitates 10^{15} calls to get the value of G_n for a single value of strength x.

B. Recursion with memoization

Although the straightforward implementation of the recursive formula (1) (given above) is elegant and close to the mathematical definition, it is not very practical. The time required to calculate $G_n(x)$ is exponential in n. To remedy this, we can employ memoization to cache previous computations. A closer look at the structure of the formula (1) reveals that many identical terms for given n and k are computer repeatedly. These terms can be stored in memoization cache and accessed for repeated use. The memoization cache is an 2D array gn_arr where one axis is accessed over n and the other one over k and the stored value is the corresponding $G_n(x)$. The memoized G_n function recursively computes and stores the value of $G_n(x)$ if it has not been previously stored in the memo array. Otherwise it returns the memoized value of G_n. The memoized values fill the lower triangle of the array ($\frac{1}{2}n(n+1)$ values). This implementation uses packages NumPy to store data in an array and mpmath to increase floating-point precision of numbers.

```
import numpy as np
import mpmath as mp

# 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.

```
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, i + 1):
            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):

```
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:

```
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.pretty = True
>>> mp.mp.dps = 30
>>> shape = mp.mpf('6.')
>>> scale = mp.mpf('1.')
>>> n_fil = mp.mpf('10')
>>> x = mp.exp(mp.mpf('-1'))
>>> binom_tab = get_binom_tab(n_fil)
>>> gn_mp(x, scale, shape, n_fil, binom_tab)
0.000000329859130502740500994574682994
'''
gn_arr = np.zeros((n, n), dtype=object)
gn_arr.fill(None)
cdf_arr = np.zeros(n, dtype=object)
cdf_arr.fill(None)
x_arr = np.zeros(n, dtype=object)
for i in range(1, int(n)):
    x_arr[n - i] = mp.fraction(n, n - i) * x
def recursion_gn_mp(x_val, scale, shape, n):
    index_n = int(n) - 1
    res = MPF_ZERO
    cdf = cdf_arr[index_n]
    if cdf == None:
        cdf = weib_cdf(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
            komb = binom_tab[index_n, k - 1]
            if k != n:
                gn = (komb * cdf_k *
                      recursion_gn_mp(x_arr[index_n],
                                      scale, shape, n - k))
            else:
                gn = komb * cdf_k # * G_0(x) (= 1.0)
            gn_arr[index_n, k - 1] = gn
        res += gn
    return res
#execute recursion
gn_m = recursion_gn_mp(x, scale, shape, n)

return gn_m
```

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

For number of fibers $n > 980$ 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 ($\frac{(n-2)^2(n-1)}{2} - \frac{(n-3)(n-2)(n-1)}{3}$ 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

$$G_n(x) = \sum_{k=1}^{n} \underbrace{(-1)^{k+1}\binom{n}{k}}_{B_{i,k}} \underbrace{[F(x)]^k}_{F_i} \underbrace{G_{n-k}\left(x\frac{n}{n-k}\right)}_{S_i}. \quad (3)$$

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 renamed 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 $x_i = x\frac{n}{i}$, $i = 1, \ldots, n$. The values of x are stored in an array named x_arr.

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

$$\boldsymbol{F}: \quad F_i = F(x_i) = F\left(x\frac{n}{i}\right), \quad i = 1, \ldots, n \quad (4)$$

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 $\boldsymbol{B}$, with n rows and n columns, pre-filled with binomial coefficients multiplied by the alternating sign. Each element of the triangular matrix initially reads

$$\boldsymbol{B}: \quad B_{i,k} = (-1)^{k+1}\binom{n}{k}, \quad i = 1, \ldots, n \text{ and } k = 1, \ldots, i. \quad (5)$$

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 $\boldsymbol{B}$ matrix):

1. Loop over n columns of the $\boldsymbol{B}$ matrix – Starting with the first column $k = 1$, each column $k = 1, \ldots, n$ is multiplied by the kth power of elements F_i: $B_{i,k} = B_{i,k}F_i^k$ for $k = 1, \ldots, n$ and $i = k, .., n$.
2. Loop over $n-1$ rows of the $\boldsymbol{B}$ 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 $G_n(x)$:

$$G_n(x) = S_n = \sum_{k=1}^{n} B_{n,k} \quad (6)$$

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 $x\frac{n}{i}$:

$$S_i = G_i(x\frac{n}{i}) = \sum_{k=1}^{i} B_{i,k} \quad i = 1, \ldots, n. \quad (7)$$

The following source code implements the described algorithm.

$$
\begin{aligned}
S_1 &= \sum \left[\; B_{1,1}F_1^1 \right. \\
S_2 &= \sum \left[\; B_{2,1}F_2^1 S_1 \quad B_{2,2}F_2^2 \right. \\
S_3 &= \sum \left[\; B_{3,1}F_3^1 S_2 \quad B_{3,2}F_3^2 S_1 \quad B_{3,3}F_3^3 \right. \\
G_4 &= \sum \left[\; B_{4,1}F_4^1 S_3 \quad B_{4,2}F_4^2 S_2 \quad B_{4,3}F_4^3 S_1 \quad B_{4,4}F_4^4 \right]
\end{aligned}
$$

Fig. 1: Diagram of the cached array for the number of fibers $n = 4$. The sum of the last line returns $G_4(x)$. (red: row indexes i, blue: column indexes k, sums of rows S_i as diagonal members)

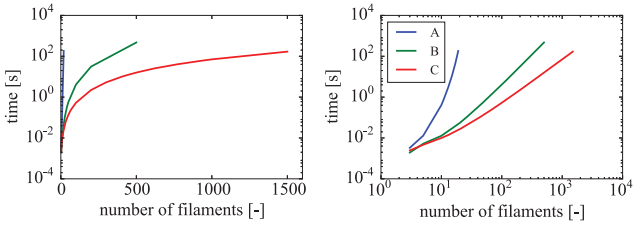

Fig. 2: Execution time of implementations A, B, and C. Left: semi-log plot shows the exponential dependence on n of the implementation A (blue linear line). Right: log-log plot.

```
def gn_mp(x, scale, shape, n, binom_tab):
    ''' see doc string in memoized version'''
    ni = int(n) # retype n from mp.mpf to int
    cdf_arr = np.zeros(ni, dtype=object)
    x_arr = np.zeros(ni, dtype=object)
    # precalculate x and F(x) vectors
    for i in range(0, ni):
        xx = mp.fraction(n, n - i) * x
        x_arr[ni - i - 1] = xx
        cdf_arr[ni - i - 1] = weib_cdf(xx, shape, scale)

    # prepare B matrix
    gn_arr = binom_tab[:ni, :ni].copy()
    for i in range(ni): # loop 1
        gn_arr[i:ni, i] *= cdf_arr[i:ni] ** (i + 1)

    for k in xrange(1, ni): # loop 2
        idx1 = np.arange(0, ni - k)
        idx2 = np.arange(k, ni)
        idx3 = np.arange(0, k)
        gn_arr[idx2, idx1] *= np.sum(gn_arr[k - 1, idx3])
    gn_m = np.sum(gn_arr[-1, :])

    return gn_m
```

IV. Discussion

Fig. 2 shows the measured execution times of the described implementations to evaluate $G_n(x)$ for a single value of load (random strength) x and for various numbers of fibers. The execution times for selected values of n are arranged in Tab. I. The values of execution times of the recursive implementation (A) were estimated on the basis of exponential dependency on n.

TABLE I
Calculated and Estimated Execution Times for Implementations of the Recursive Formula.

n	A	B	C
50	8094 years	0.58 s	0.12 s
1000	$2.5 \cdot 10^{284}$ years	1 h	70 s
1500	—	3.5 h	167 s

[Dell T7610, 2x Intel Xeon E5-2687w, 192 GB, Kubuntu 14.04]

The implementations using mpmath enable to calculate values of CDF for large numbers (thousands) of fibers in bundle including values deep in the left tail of the distribution (probabilities 10^{-600}). In addition, the loop-based implementation C reduces significantly the execution time (speedup for $n = 1000$ compared to B is $51\times$ and for $n = 1500$ it is $75\times$).

V. Analysis of Addends

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 $\sigma_x = 0.180$).

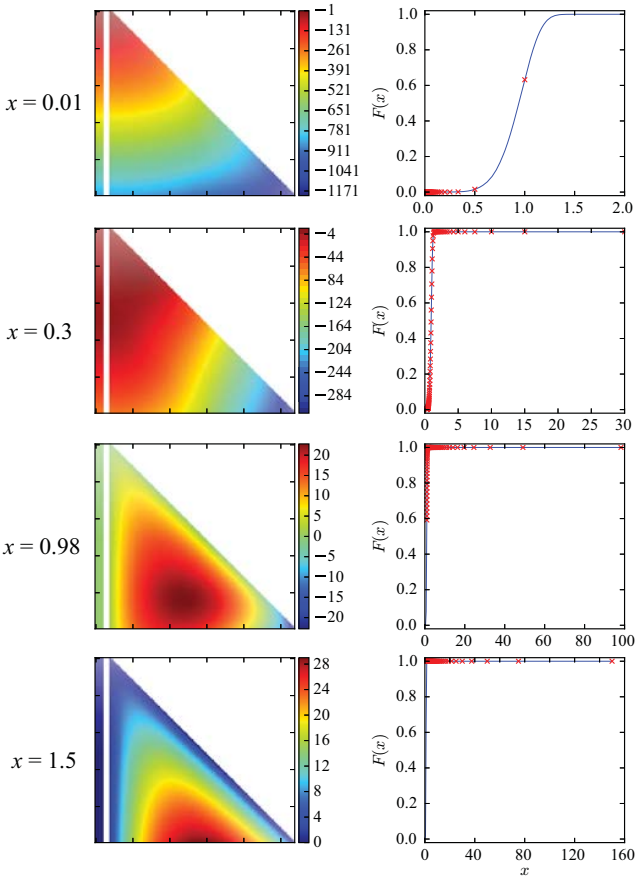

Fig. 3: Left: left column plots orders of the sum of the row and triangle contains orders of the $\boldsymbol{B}$ matrix. Right: CDF plot (blue line) and values of x and $F(x)$ used in evaluation of G_n (red crosses)

Fig. 3 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 $B_{i,k}F_i^k S_i \approx B_{i,k} \cdot 1^k \cdot 1 = B_{i,k}$.

Based on the analysis of contributions to the value of G_n from the $\boldsymbol{B}$ matrix, we conclude that no systematic way of ignoring any elements of the $\boldsymbol{B}$ 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

[1] H.E. Daniels, The Statistical Theory of the Strength of Bundles of Threads, *Proc. Royal Soc. (London)*, 183A, 405–435, 1945.
[2] H.E. Daniels, The Maximum of a Gaussian Process Whose Mean Path Has a Maximum, with an Application to the Strength of Bundles of fibers, *Advances in Applied Probability*, 21 (2), 315–333, 1989.
[3] R.L. Smith, S.L. Phoenix, Asymptotic Distributions for the Failure of Fibrous Materials Under Series-Parallel Structure and Equal Load-Sharing, *Journal of Applied Mechanics*, 48, 75–82, 1981.
[4] D. Sornette, Elasticity and failure of a set of elements loaded in parallel., *J. Phys. A: Math. Gen.*, 22, L243–L250, 1989. (Letter to the Editor).
[5] G. Rossum, et al., The Python Programming Language, http://python.org, 1991.
[6] F. Johansson, et al., mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18), http://mpmath.org, 2014.
[7] T. Oliphant, et al., NumPy: Open source scientific package for scientific computing with Python, http://numpy.scipy.org, 2006.