ELSEVIER

Computer Physics Communications 98 (1996) 224–234

Computer Physics Communications

Implementation of a fast box-counting algorithm

A. Kruger [1]

404 HL, Iowa Institute of Hydraulic Research, The University of Iowa, Iowa City, IA 52242, USA

Received 8 March 1996; revised 26 March 1996

Abstract

We briefly summarize a previously published fast box-counting algorithm for computing the box-count fractal dimension, and adapt it to provide more points on the box-count versus box-size curve. We then present a C language implementation of the algorithm that consists of a routine called `boxcount`. It uses a flexible memory-addressing scheme that makes it suitable for analyzing multidimensional data, as well as analyzing time-series data efficiently with different embedding dimensions. A stand-alone program that uses `boxcount` is also presented.

Keywords: Fractal; Fractal dimension; Nonlinear dynamics; Box-counting; Bit-interleaving

PROGRAM SUMMARY

Title of program: boxcount and boxdim

Catalogue identifier: ADDW

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland

Licensing provisions: none

Computers for which the program is designed and others on which it is operable:
Computers: Any system with an ANSI-compliant C compiler; *Installations:* Hewlett-Packard series 9000, IBM RS/6000, Silicon Graphics IRIS, Intel x86

Operating systems under which the program has been tested: HP-UX, IRIX, AIX/6000, MS-DOS

Programming Language used: ANSI C

Memory to execute with typical data: About 16MB for 1 million data points embedded in 2-dimensional space

Number of bits in a word: 16 or more

Has the code been vectorized? No

Number of bytes in distributed program, including test data: etc.: 1379835

Distribution format: ASCII

Keywords: fractal, fractal dimension, nonlinear dynamics, box-counting, bit-interleaving

Nature of physical problem
Numerical estimation of the fractal dimension of time series and multidimensional data.

Method of solution
A fast box-counting algorithm based on bit-interleaving of data points.

Restrictions on the complexity of the problem
Machine memory.

Typical running time
About 82 sec on a 97 MIPS, 28 MFLOPS machine for a 1 million point time series embedded in 2-dimensional space.

[1] E-mail: kruger@iihr.uiowa.edu

PII S0010-4655(96)00080-X

Unusual features of the program
Very fast, easy to analyze multidimensional data, time-series data, and time-series data with different delay embeddings.

LONG WRITE-UP

1. Fractal dimension from box counts

The capacity or box-counting dimension is defined as follows. Overlay the p-dimensional space that embeds the data points with a fixed-size grid of p-dimensional boxes with sides r, and count the number of nonempty boxes n. The capacity dimension is then

$$D_c = \lim_{r\to 0} \frac{\log(1/n(r))}{\log r} = -\lim_{r\to 0} \frac{\log n(r)}{\log r}. \quad (1)$$

The base of the logarithm in Eq. (1) is in principle not important. However, a geometric interpretation of the box-count algorithm implemented in this paper is that the p-dimensional space is overlaid with a sequence of boxes with sides that double on each pass through the algorithm. This makes it convenient to use 2 as the base of the logarithm, since values on the $\log r$-axis are then equally spaced. Thus, in this paper the implied base is 2, and except for the figure axis labels, the base is not shown.

Some factors related to box-counting algorithms are discussed briefly here in terms of Fig. 1.

Finite resolution

In general, only a finite computing resolution is available and one cannot let $r \to 0$ in Eq. (1). One common approach is to plot $\log(n(r))$ against $\log r$ and use the slope of this curve to approximate D_c,

$$D_c \approx -\frac{\Delta \log(n(r))}{\Delta \log r}. \quad (2)$$

Finite N

The number of data points N available is finite, limited either by experimental factors, or by space requirements. Following Theiler's notation [1], let $\hat{n}(N,r)$ be the number of nonempty boxes. $\hat{n}(N,r)$ is bounded by $n(r)$, but for fixed r approaches $n(r)$,

$$\hat{n}(N,r) < n(r), \quad (3)$$

$$\lim_{N\to\infty} \hat{n}(N,r) = n(r). \quad (4)$$

This is shown in Fig. 1a.

As r is decreased, a point is reached where each data point is covered with a single box and $\hat{n}(N,r)$ is fixed for smaller r,

$$\hat{n}(N,r) = N, \quad r \le r_{\min}. \quad (5)$$

which leads to what we refer to as the *finite N plateau* in the $\log \hat{n}(N,r)$ curve. This is shown in Fig. 1b. The practical implication of the finite N plateau is that the number of points on the slope of the $\log \hat{n}(N,r)$ versus $\log r$ curve is small. This in turn influences the accuracy of the slope determination and thus the estimation of the fractal dimension.

Finite size and edges

Fig. 1c shows the Hénon attractor covered completely with a box, so $\hat{n}(N,r_1) = 1$. Boxes bigger than box A will also enclose the attractor completely, and

$$\hat{n}(N,r) = 1, \quad r \ge r_1, \quad (6)$$

that leads to the *finite size plateau* in the $\log \hat{n}(N,r)$ curve. This is shown in Fig. 1e. When a box is slightly smaller than box A, such as the boxes in Fig. 1d, two boxes are needed to cover the attractor completely. In fact, there are several pairs of boxes smaller than box A that will cover the attractor completely, and

$$\hat{n}(N,r) = 2, \quad r_2 \le r < r_1. \quad (7)$$

Here r_2 is the smallest r that still allows coverage of the attractor with two boxes. When the boxes are made smaller still, a point is reached where three boxes are needed to cover the attractor. As in the previous cases, several differently-sizes boxes will do, and

$$\hat{n}(N,r) = 3, \quad r_3 \le r < r_2, \quad (8)$$

and so on. For these rs, which are large compared to the overall dimensions of the attractor, $\hat{n}(N,r)$ is fixed for ranges of r, and when $\log \hat{n}(N,r)$ is plotted against $\log r$, this staircasing is clear. As r becomes smaller, the staircasing becomes less prominent and eventually $\log \hat{n}(N,r)$ versus $\log(r)$ becomes a smooth curve, see Fig. 1e.

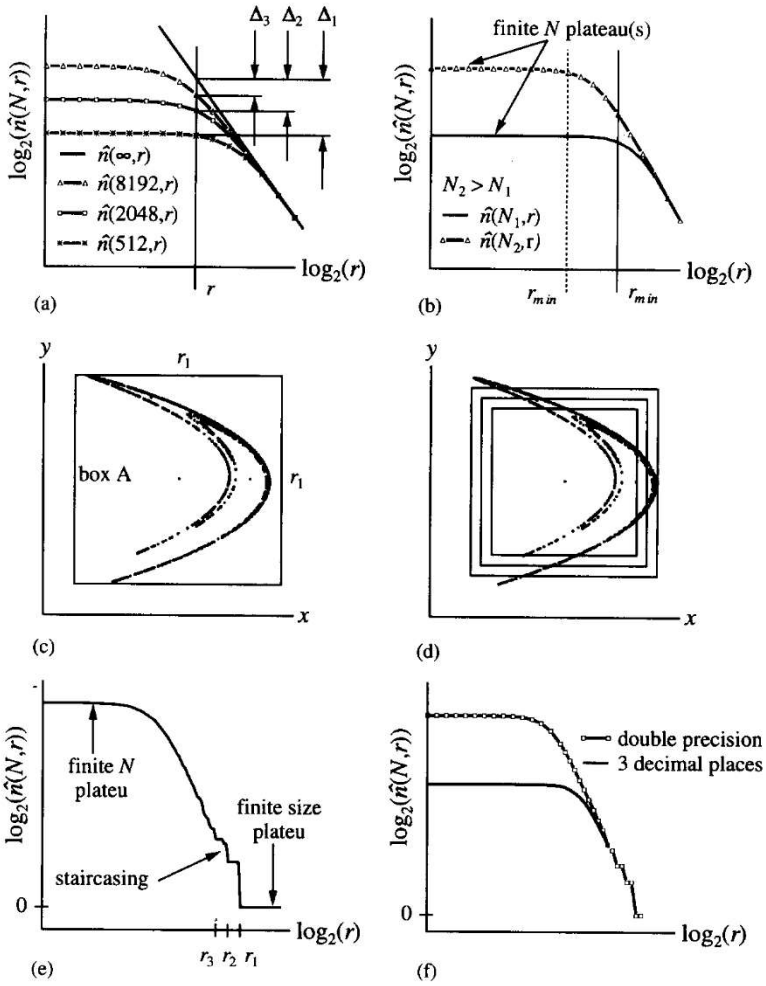

Fig. 1. Practical issues related to box-counting algorithms. (a) For fixed r, $\hat{n}(N,r)$ is less than $n(r)$, but the difference between $n(r)$ and $\hat{n}(N,r)$ goes to 0 as N is increased. (b) When the box sides are smaller than $r_{\min}$ each data point is covered with a box – decreasing r does not lead to an increase in the number of boxes needed to cover the data. However, increasing N shifts $r_{\min}$ and leads to more points on the slope. (c) Smallest box that will cover data. (d) Boxes smaller than box A. Two of these are needed to cover the data. (e) At large r, edge effects manifest as staircasing. (f) Quantization of the data leads to a lowering of the intercept with the $\log_2 \hat{n}(N,r)$ axis. This implies fewer points on the slope.

Quantization

In order to process analog signals on a computer they must be sampled, quantized, and converted to series of numbers. When the correlation integral is used to compute the fractal dimension of such a quantized series, the quantization results in staircasing at small r [1]. In the case of the box-counting algorithm it leads to a lowering of the intercept of the curve with the $\log \hat{n}(N,r)$ axis, which affects the number of points on the curve that can be used to determine D_c.

Quantization is related to the issue of finite resolution discussed above. However, the effects of quantization are several orders of magnitude more pronounced. Most analog to digital converters are 12- or 16-bit, so that signals are quantized to 4096 or 65 536 levels. On a typical IEEE-compliant computer the (single) floating-point precision is 1.19×10^{-7}, which translates to $1/1.19 \times 10^{-7} = 8.4 \times 10^{6}$ "quantization levels".

Quantization effects are illustrated in Fig. 1f where

two curves for $\log \hat{n}(N,r)$ versus $\log r$ (for the Hénon map) obtained via a box-counting algorithm are shown. One curve is for the case where double precision computations were employed where applicable. The other curve is for the case where computations where done to 3 decimal digits, thus simulating quantization.

2. Box-counting via bit-interleaving

An efficient algorithm for box-counting was published by Hou et al. [2]. The algorithm was mentioned in an earlier paper [3] and is similar to work documented elsewhere [4]. Briefly, the algorithm is based on the concept of *bit-intercalation* or *bit-interleaving* [5], obtained by first mapping the p coordinates of the p-dimensional input data to the range 0–$2^m - 1$, where m is an integer. Then the m bits from each coordinate are bit-interleaved to form a bit string. Because the bit string uniquely determines the position of the data point in p-dimensional space, we refer to it as the *position code* for the data point. The position codes are gathered in a table and sorted. The number of unique entries in the position code table, while ignoring ip bits, is counted by scanning the position code table and noting the number of changes. This is repeated for $i = 0, 1, \ldots, m$, and this process is equivalent to overlaying the p-dimensional space with a sequence of $m+1$ p-dimensional boxes with sides $r = 1, 2, \ldots, 2^m$, from which the box-count information follows. The computational complexity of the algorithm is $O(pN \log N)$.

3. Increasing points on slope

The basic algorithm gives a sequence of $m+1$ box counts for boxes with sides $2^0, 2^1, \ldots, 2^m$. In other words, the sides of the boxes are doubled on each pass through the algorithm. This may not be the optimum set of boxes, and depending on the data set, most points on the $\log \hat{n}(N,r)$ curve may lie on the finite N plateau, on the finite size plateau, or on both plateaus, with only a few points on the slope. This means that only a few points can be used to determine D_c from Eq. (2). A little thought will show that the first step of the algorithm, namely mapping the coordinates of the p-dimensional data so that they lie in the range 0–$2^m - 1$, can be used to solve this problem. If, for example, the data is mapped so that the coordinate range is K, where $K < 2^m - 1$, and the algorithm is applied, the integer-mapped set is again overlaid with sets of boxes whose sides double on each iteration. However, the extent of the integer-mapped coordinates are now smaller by a factor $(2^m - 1)/K$. Relative to the integer-mapped coordinates, the boxes are bigger by a factor $K/(2^m - 1)$ and the algorithm will give a new set of points on the $\log \hat{n}(N,r)$ curve.

The process can be repeated, say N_s times, and the sequence

$$K = 2^{l/N_s}, \quad l = 0, \ldots, N_s - 1, \tag{9}$$

ensures that the points on the $\log \hat{n}(N,r)$ curve are equally spaced.

4. Implementing position codes

The portable range of unsigned integers in C determines m. In C, used portably, unsigned long variables are at least 4 bytes or 32 bits long [6] so in the implementation described here, we have used $m = 32$. For p-dimensional data with coordinates that lie in the range 0–$2^m - 1$, each position code is mp bits long, and this can lead to overflow problems. For example, with $m = 32$ and $p = 3$ the position codes are 96 bits long – longer than any scalar variable on most machines. A solution is to break the position codes up into smaller sections and store these sections in an integer array. For the sake of space efficiency, these smaller sections should be the size of the smallest scalar type available in the implementation language.

This is typically 8 bits, but as an illustration, consider the 6-bit position codes,

111001_2

100111_2

100110_2

100101_2

011111_2

010010_2

001010_2

001001_2

and break them up into 2-bit sections. Position code 111001_2 becomes 11_2, 10_2, and 01_2, or 3, 2, and 1. Similarly, position code 100111_2 becomes 2, 1, and 3, position code 100110_2 becomes 2, 1, and 2, and so on. In a computer program these values can be stored in an 8×3 array, and each row of the array corresponds to a position code,

$$\begin{array}{l} \text{position code for } 111001_2 \longrightarrow \\ \text{position code for } 100111_2 \longrightarrow \\ \qquad\vdots \end{array} \begin{bmatrix} 1\,2\,3 \\ 3\,1\,2 \\ 2\,1\,2 \\ 1\,1\,2 \\ 3\,3\,1 \\ 2\,0\,1 \\ 2\,2\,0 \\ 1\,2\,0 \end{bmatrix}$$

The order of the sections for a position code in each row is not important, as long as one is consistent. For coding convenience we choose the order shown above: the least significant sections are in lower memory locations.

Manipulating the position codes (sorting, comparing, copying) is then done the same way one would manipulate strings: one character/byte at a time. One clean and simple approach to managing the position code table is to dynamically allocate memory for the position codes as they are created and access them through an array of pointers that is also allocated dynamically. Manipulating the position codes via pointers reduces the overhead that goes with moving the position codes themselves during the sort phase of the algorithm. Another possibility is to build a binary search tree of the position codes as they are constructed.

However, for reasons of efficiency and other considerations such as memory alignment, operating systems often allocate a minimum number of bytes regardless the requested number of bytes. When a program makes only a few memory allocation requests the memory wasted in this fashion is seldom a problem. On the other hand, when a program makes hundreds of thousands and possibly millions of memory allocation requests, as is potentially the case with `boxcount`, this wasted memory can quickly become significant.

An alternative strategy, which we have used in `boxcount`, is to allocate the memory required for the all position codes as a large array of 1-byte integers, say *codes*. Since the position codes are all N_{bytes} long, they are easily accessed by indexing *codes* appropriately. For example, the position codes are numbered $0, 1, \ldots$, and position code j is located at $codes[j * N_{\text{bytes}}]$. This approach reduces the memory requirements by eliminating the table of pointers (as well as the wasted space associated with each pointer), but it implies that the position codes are moved when the position code table is sorted. On the other hand, instead of a memory allocation request for each position code, there is now only one request for the whole table.

5. Sorting position codes

Sorting the position code table is easily accomplished with the standard C library function `qsort`, which is an implementation of the Quicksort algorithm. In addition to an array that must be sorted, one must supply `qsort` with a suitable routine for comparing two entries in the array. As we have noted earlier, if one breaks up a long position code into smaller sections and store each section in consecutive memory locations, the position codes can be manipulated much the same way one would manipulate strings. For example, if `*pc1` and `*pc2` are the starting addresses of two position codes and `nbytes` (a global variable) their length, then code such as

```
static int bytecmp(byte *pc1, byte *pc2)
{
 unsigned long i,indx;

 for (i=1L;i<=nbytes;i++){
  indx = nbytes-i;
  if (pc1[indx] < pc2[indx]) return (1);
  if (pc1[indx] > pc2[indx]) return (-1);
 }
 return(0);
}
```

can be used as the compare routine for `qsort`. The reverse-order comparison is a side effect of how we build the position codes – least significant parts are stored in lower memory locations.

A note of caution is in order with respect to using `qsort`. When properly implemented, Quicksort is the fastest sorting algorithm for many sorting problems. In textbooks it is often introduced and first implemented

as a recursive algorithm, and then reimplemented as an iterative algorithm. However, we have encountered at least one widely-used commercial C compiler that implements `qsort` as a recursive algorithm. Not only is it less efficient than what it should be, but when sorting large arrays, the program's stack space is depleted! If the input data is already sorted or almost sorted, Quicksort's performance degenerates, and a solid implementation of Quicksort accounts for this by scrambling the input data a little bit before sorting it. It is admittedly unlikely that the position code table will initially be sorted or almost sorted, considering that the position codes are constructed by interleaving bits from several numbers. Nonetheless, unless one has full confidence in the `qsort` implementation on a particular system, it may be prudent to supply one's own sort algorithm. The software distribution described in this article contains a fairly efficient implementation of the Heapsort algorithm that may be used in lieu of `qsort`, and the compile "Makefile" gives the user the option to choose either one.

6. Time series: delay-time embedding

The usual method for reconstructing the phase space from a time series is Packard's delay scheme [7] where the time series $x(t_0), x(t_1), x(t_2), \ldots$, is embedded into p-dimensional space by creating vectors

$$\begin{aligned} \boldsymbol{x}_0(t) &= [x(t_0), x(t_0-\tau), \ldots, x(t_0-(p-1)\tau)], \\ \boldsymbol{x}_1(t) &= [x(t_1), x(t_1-\tau), \ldots, x(t_1-(p-1)\tau)], \\ &\vdots \\ \boldsymbol{x}_{N-1}(t) &= [x(t_{N-1}), x(t_{N-1}-\tau), \ldots, \\ &\qquad x(t_{N-1}-(p-1)\tau)]. \end{aligned} \tag{10}$$

Here p is called the embedding dimension, τ is the delay, and N is the number of integral vectors that can be constructed from the time series in this fashion. An example for a 9-point time series is shown in Fig. 2a–c.

It can be shown that the procedure almost always reconstructs the phase space as long as $p > 2D + 1$, where D is the fractal dimension. Also, if $p > D$, the reconstructed set will have the same fractal dimension as the attractor, and if $p > D$, most values of τ can be used to reconstruct the phase space. However, finding

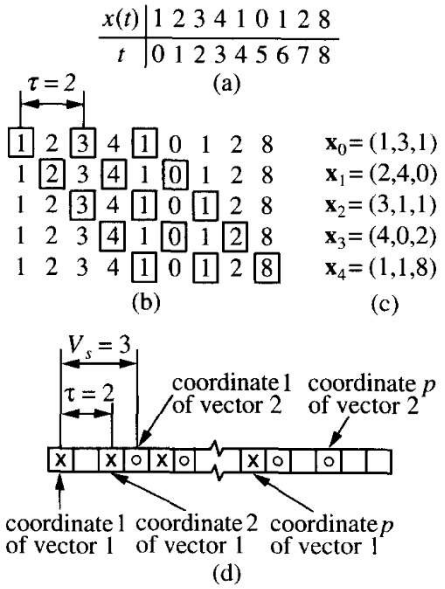

Fig. 2. Delay time embedding example for $p = 3$ and $\tau = 2$. (a) A 9-point time series. (b) Individual components of each reconstructed vector are marked with a box. (c) The corresponding array of vectors. In a computer program, one can store the array of vectors as a 5×3 matrix of numbers. (d) V_s is the spacing between the start of two consecutive vectors and τ is the spacing between the individual components of a vector.

an optimal τ is not so easy, and several heuristics have been suggested.

The general idea is to have both τ and $(p-1)\tau$ close to some characteristic decorrelation time. A common approach is to use the delay τ that will result in

$$\rho(\tau) = 0.5, \tag{11}$$

where ρ is the linear autocorrelation function of the time series. Another approach is to use the first minimum in the mutual information of the time series [8]. Once a delay has been chosen, the fractal dimension D is computed for increasing embedding dimension p, until an increase in p does not lead to a significant change in D. This saturation value of D is then taken as the fractal dimension. Some authors have suggested keeping $(p-1)\tau$ constant as p is increased [9,10].

It is clear that it is desirable to be able to easily experiment with different values for p and τ. One possibility is to construct an array of vectors for a given p, τ that becomes the input for a program/subroutine that computes the fractal dimension. In a computer program, the array of vectors can be constructed by

copying the appropriate values from the time series into an $N \times p$ matrix. Ideally, however, one should not have to reorganize the time-series data for different values of p and τ, since this can be quite inefficient in terms of space as well as running time. Rather, the algorithm that is used to compute the fractal dimension should index the time series appropriately. The routine `boxcount` described below uses this approach.

It follows from Fig. 2b that with $p = 3$ and $\tau = 2$, one can construct $N = 5$ vectors from a 9-point series. An expression for N in terms of p, τ, and the number of time-series data points will now be derived. Let the time series be stored in a linear array, and construct p-dimensional vectors with a delay τ and vector stride V_s. An example is shown in Fig. 2d. If the array is indexed 0,1,..., then vector k starts at index $(k-1)V_s$. Also, each vector spans

$$p + (p-1)(\tau - 1) = (p-1)\tau + 1 \qquad (12)$$

locations, and vector k ends at index

$$((k-1)V_s) + ((p-1)\tau + 1) - 1 = (k-1)V_s + (p-1)\tau . \qquad (13)$$

Thus, the pth coordinate of the kth vector is the N_tth point of the time series, where

$$N_t = (k-1)V_s + (p-1)\tau + 1 . \qquad (14)$$

Given an array of N_t values, how many vectors N with dimension p, delay τ, and vector stride V_s can one construct? From the equation above one can solve for k,

$$k = (N_t - 1 + V_s - (p-1)\tau)/V_s , \qquad (15)$$

that is in general nonintegral. Thus the integral number of vectors is given by

$$N = \lfloor k \rfloor = \lfloor (N_t - 1 + V_s - (p-1)\tau)/V_s \rfloor , \qquad (16)$$

where $\lfloor\ \rfloor$ is the "floor" operator. That is, $\lfloor x \rfloor$ returns the largest integer not greater than x.

7. A C routine for box-counting: boxcount

The C routine for computing box counts, `boxcount`, is called with several parameters that determine the interpretation of the data points,

```
#include <boxcount.h>
        .
        .
        .
boxcount(x,nt,p,tau,vs,n,max,min,
         k,rs,counts);
```

where the floating-point array `x` contains the time-series data and `nt` is the number of data points in the time series. The parameters `p`, `tau`, and `vs` determine the delay-time embedding. The parameter `n` is the number of p-dimensional vectors that must be reconstructed and its maximum value is given by Eq. (16). The floating-point parameters `max` and `min` are the maximum and minimum values of the time-series data. The floating-point parameter `k` is the integer-mapping range factor, see Eq. (9). The return parameter `rs` is an array of $m + 1 = 33$ box sizes $r = 1, 2, \ldots, 2^{32}$ and the return parameter `counts` is an array that holds the 33 corresponding box counts. All the parameters are unsigned long except where otherwise noted.

The routine `boxcount` calls several lower-level routines for performing some of the algorithm steps such as bit-interleaving, sorting, and counting. These routines are short, amply commented, and will not be described here. All the routines are in a single source file `boxdim.c` and there is a single header file `boxcount.h`.

To analyze a large data set, one would read or compute the time-series data and keep it in RAM, and then call `boxcount` with different values for the embedding dimension and/or delay. On a first pass through the data, one would typically set the integer-mapping range factor `k` equal to 1 and plot the logarithm of `count` versus the logarithm of `r`. If more points on the slope are desired, then one can call `boxcount` with several values of `k`, see Eq. (9). The stand-alone program described in the next section is a good example of how to use the routine `boxcount`, and users can model their own programs after this program.

8. A stand-alone program for box-counting: boxdim

The program `boxdim` is a stand-alone program that calls `boxcount` to compute box-count information. It takes two arguments on the command line. The first argument is the name of the input data file that may

contain real or integer data points. The file structure is ignored – data points are treated as consecutive points of a time series and must be separated by at least one space, tab, or newline character. For example, a file with contents

```
1 2 3 4 1 0 1 2 8
```

conveys the same information to `boxdim` as a file with contents

```
1 2 3
4 1 0
1 2 8
```

The second command line argument is the name of a *parameter* file that contains the analysis parameters N_t, p, τ, V_s, and N_s. This is simply a file that contains optional comment lines (lines that start with the # character) and the analysis parameters separated by blanks or tabs. For example,

```
# Nt        p   tau  Vs   Ns
  4194304   2   1    2    2
```

instructs `boxdim` to read 4 194 304 data points from the input file, use an embedding dimension of 2, a delay of 1, and a vector spacing of 2 when reconstructing the phase space, and run the algorithm $N_s = 2$ times to increase the number of points on the slope. The output consists of two columns of real numbers that are printed on the standard output. The first column is $\log_2 r$, and the second is $\log_2 \hat{n}(N, r)$. The user can plot and analyze the columns to extract fractal dimensions. The power of this approach is that in order to experiment with different values of embedding dimension and delay, one simply has to change the parameter file – there is no need to reformat the data file.

One can use `boxdim` to analyze time-series data as well as true multidimensional data. To analyze multidimensional data, view it as a time series, and set the analysis parameters appropriately. Consider, for example, 5 vectors $(1,3,1)$, $(2,4,0)$, $(3,1,1)$, $(4,0,2)$, and $(1,1,8)$. To use `boxdim` with this data, one could store it in a file as follows:

```
1 3 1
2 4 0
3 1 1
4 0 2
1 1 8
```

and create a parameter file,

```
# Nt   p  tau  Vs  Ns
  15   3  1    3   3
```

When `boxcount` reads this data file it treats the data as a 15-point time series,

$x(t)$	1	3	1	2	4	0	3	1	1	4	0	2	1	1	8
t	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

It reads $N_t = 15$, $p = 3$, $\tau = 1$, $V_s = 3$, from the parameter file, constructs the sequence of $N = 5$ (see Eq. (16)) vectors $(1,3,1)$, $(2,4,0)$, ..., and runs the box-counting algorithm $N_s = 3$ times on the data.

9. Results

The program `boxdim` was applied to computed data for several dynamical systems, and the results are shown in Fig. 3. The running time for `boxdim` on a 97 MIPS, 28 MFLOPS workstation when applied to a 1-million point time series embedded in 2-dimensional space is about 163 seconds. Fig. 4 shows a breakdown of the measured running time. About 50% of the running time comes from reading the data. The program read the data from a local disk, so the reading time was not influenced by network traffic. Also, the workstation used for the measurements was equipped with enough RAM to satisfy all the memory requirements, and virtual memory swapping did not contribute substantially to the running time. Most of the remaining running time comes from bit-interleaving, sorting, and counting. The total running time is about 163 seconds so the box-counting time is about 82 second.

Measurements made on smaller and larger time series, different machines, and with different compilers on these machines do not change the relative size of these numbers significantly. The fact that about half of the running time comes from reading the time series shows the efficiency of the bit-interleaving algorithm. Furthermore, the routines that perform the bit-interleaving, sorting, and counting phases, each contribute significantly to the running time, so leaving the reading time aside, the implementation of the algorithm is well balanced. A Heapsort routine was used for these measurements – see the comments at the end of Section 5 on which sort routine to use.

As it stands, `boxcount/boxdim` is very efficient and should suffice in most situations, even when large time series are read from external storage. However,

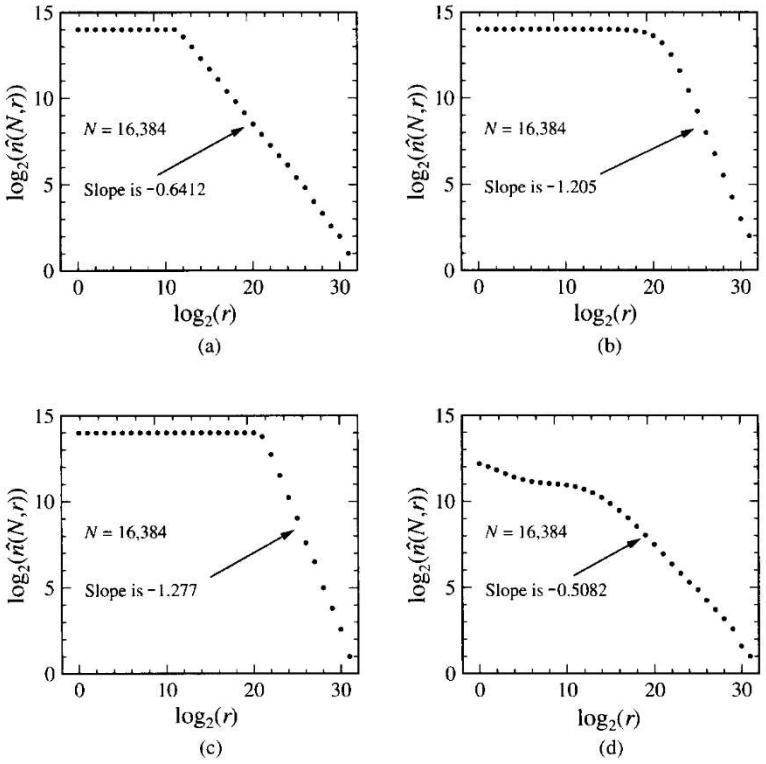

Fig. 3. Box-count results for several well-known dynamical systems obtained with the program `boxdim` described in the text. (a) Cantor Map. (b) Hénon Map. (c) Koch Curve. (d) Logistic Map.

if one wants to decrease the running time an option is to merge some of the lower-level routines that `boxcount` calls with `boxcount`. This will eliminate the overhead of many function calls, but this reduces program modularity and clarity. For programs such as `boxdim`, that may read massive amounts of data, and then call `boxcount` to get box-count information, it makes more sense to speed up reading the data. One possibility is to keep the time-series data in binary format that is often much faster to read than ASCII-formatted data.

The standard C input–output library is known to be less than optimum in terms of efficiency, and the *sfio* library [11] can, in certain situations, dramatically (2–4 times faster) reduce input–output times. Since it provides the same calling and naming conventions as the standard C input–output library, no source level changes to a program are necessary. Thus, the most sensible approach to reducing the running time of programs such as `boxdim` is to use the *sfio* library.

10. Portability

Great care was taken to write the routine `boxcount` and the stand-alone program `boxdim` program in portable, ANSI-compliant C. Both were compiled and tested on Hewlet-Packard series 9000, IBM RS/6000, and Silicon Graphics IRIS workstations, as well as on Intel x86-based computers. On the workstations, the native C compilers as well as the GNU C compiler were used, and on the Intel x86-based computers, Borland's and Microsoft's compilers (several versions) were used. There are some obscure options of the GNU C compiler that produce a few warnings, but these may safely be ignored. Both `boxcount` and `boxdim` pass the UNIX *lint(1)* program.

11. Test programs

In addition to the software described so far, the distribution contains a file `koch.dat` that contains 16 384

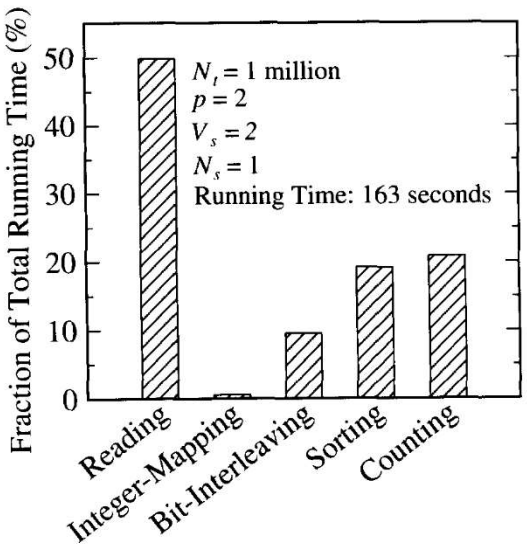

Fig. 4. A breakdown of boxdim's running time. Measurement were made on an 97 MIPS, 28 MFLOPS, Hewlett-Packard series 9000 machine. The data was read from a disk local to the machine. Sorting was done with the Heapsort routine (see text).

Table 1
Files in the distribution

File	Description
Makefile	UNIX Makefile for compiling and testing the stand-alone program boxdim.
boxcount.c	Contains boxcount and other C routines that implement the fast box-counting algorithm.
boxcount.h	A header file for programs that use boxcount.
boxdim.c	A stand-alone program that uses boxcount for computing fractal dimensions. An example of how to use boxcount.
koch.dat	Contains 16384 points on the Koch curve that are used for testing boxdim.
koch.par	A file that contains analysis parameters. It is used by the Makefile when testing boxdim.
Ref.out	The correct output from boxdim.

points on the Koch curve, a parameter file `koch.par`, sample output in a file `sample.out`, and an UNIX Makefile. The contents of the distribution are listed in Table 1. The Makefile builds the program `boxdim`, and executes it with input file `koch.dat` and parameter file `koch.par`. The output from `boxdim` is sent to a file `test.out` that is compared to the reference output, `Ref.out`. Next is a sample run on an HP-UX machine:

```
r-hml004% make
Usage: make machine, where "machine" is one
of:

  aix:   AIX/R6000 machines
  gcc:   GNU C compiler
  hpux:  HP-UX machines
  sgi:   Silicon Graphics machines

r-hml004% make hpux
     ./boxdim koch.dat koch.par > test.out
Good! passed test.
```

The string `r-hml004%` above is the command line prompt. Below are the contents of the file `test.out` that contains the box-count information for 16384 points of the Koch curve:

```
--------------------
log2(r) log2(n(N,r))
--------------------
 0.0000  13.0000
 0.5000  13.0000
 1.0000  13.0000
 1.5000  13.0000
 2.0000  13.0000
 2.5000  13.0000
 3.0000  13.0000
 3.5000  13.0000
 4.0000  13.0000
 4.5000  13.0000
 5.0000  13.0000
 5.5000  13.0000
 6.0000  13.0000
 6.5000  13.0000
 7.0000  13.0000
 7.5000  13.0000
 8.0000  13.0000
 8.5000  13.0000
 9.0000  13.0000
 9.5000  13.0000
10.0000  13.0000
10.5000  13.0000
11.0000  13.0000
11.5000  13.0000
12.0000  13.0000
12.5000  13.0000
13.0000  13.0000
13.5000  13.0000
14.0000  13.0000
```

```
14.5000  13.0000
15.0000  13.0000
15.5000  13.0000
16.0000  13.0000
16.5000  13.0000
17.0000  13.0000
17.5000  13.0000
18.0000  13.0000
18.5000  13.0000
19.0000  13.0000
19.5000  13.0000
20.0000  13.0000
20.5000  13.0000
21.0000  13.0000
21.5000  13.0000
22.0000  12.7731
22.5000  12.3208
23.0000  11.7343
23.5000  11.1510
24.0000  10.5196
24.5000   9.9129
25.0000   9.2167
25.5000   8.6760
26.0000   8.0224
26.5000   7.4998
27.0000   6.5850
27.5000   6.3219
28.0000   5.4919
28.5000   5.0000
29.0000   4.0000
29.5000   3.7004
30.0000   2.8074
30.5000   2.8074
31.0000   1.5850
31.5000   1.5850
32.0000   0.0000
32.5000   0.0000
```

12. Summary

We have described the implementation of a very efficient box-counting algorithm that is based on the concept of bit-interleaving. The algorithm is fast enough to make computing fractal dimensions of large time series (several million points) practicable. The algorithm implementation uses a flexible memory addressing scheme that makes it unnecessary to reformat the time series when different delay embeddings are used to reconstruct the phase space. This makes it convenient to experiment, and also makes it possible to compute box-count information for time series, as well as true multidimensional data. A stand-alone program that employs the box-counting routine was also described. It reads data and analysis parameters from files and prints box-count information to the standard output.

References

[1] J. Theiler, J. Opt. Soc. Am. 7A (1990) 1055.
[2] X. Hou, R. Gilmore, G.B. Mindlin and H.G. Solari, Phys. Lett. A 151 (1990) 43.
[3] L.S. Libovitch and T. Toth, Phys. Lett. A 141 (1989) 386.
[4] A. Block, W. von Bloh and H.J. Schellnhuber, Phys. Rev. A 42 (1990) 42.
[5] H. Samet, The Design and Analysis of Spatial Data Structures (Addison-Wesley, Reading, MA, 1990).
[6] H. Rabinowitz and C. Schaap, Portable C (Prentice Hall, Englewood Cliffs, NJ, 1990) p. 79.
[7] N.H. Packard, J.P. Crutchfield, J.D. Farmer and R.S. Shaw, Phys. Rev. Lett. 45 (1980) 712.
[8] A.M. Fraser and H.L. Swinney, Phys. Rev. 33 (1986) 1134.
[9] D.S. Broomhead and G.P. King, Physica 20D (1986) 217.
[10] S. Sato, M. Sano and Y. Sawada, Prog. Theor. Phys. 77 (1987) 1.
[11] D.G. Korn and K.P. Vo, Proc. 1991 Summer USENIX Conference (1991) 235.