3. Algorithm animation

I hear and I forget, I see and I remember, I do and I understand.

A picture is worth a thousand words—the art of presenting information in visual form.

Learning objectives:

- adding animation code to a program
- examples of algorithm snapshots

Computer-driven visualization: characteristics and techniques

The computer-driven graphics screen is a powerful new communications medium; indeed, it is the only two-way mass communications medium we know. Other mass communications media–the printed e.g. recorded audio and video—are one-way streets suitable for delivering a monolog. The unique strength of our new medium is interactive presentation of information. Ideally, the viewer drives the presentation, not just by pushing a start button and turning a channel selector, but controls the presentation at every step. He controls the flow not only with commands such as "faster", "slower", "repeat", "skip", "play this backwards", but more important, with a barrage of "what if?" questions. What if the area of this triangle becomes zero? What if we double the load on this beam? What if world population grows a bit faster? This powerful new medium challenges us to use it well.

When using any medium, we must ask: What can it do well, and what does it do poorly? The computer-driven screen is ideally suited for rapid and accurate display of information that can be deduced from large amounts of data by means of straightforward algorithms and lengthy computation. It can do so in response to a variety of user inputs as long as this variety is contained in an algorithmically tractable, narrow domain of discourse. It is not adept at tasks that require judgment, experience, or insight. By comparison, a speaker at the blackboard is slow and inaccurate and can only call upon small amounts of data and tiny computations; we hope she makes up for this technical shortcoming by good judgment, teaching experience, and insight into the subject. By way of another comparison, books and films may accurately and rapidly present results based on much data and computation, but they lack the ability to react to a user's input.

Algorithm animation, the technique of displaying the state of programs in execution, is ideally suited for presentation on a graphics screen. There is a need for this type of computation, and there are techniques for producing them. The reasons for animating programs in execution fall into two major categories, which we label *checking* and *exploring*.

Checking

To understand an algorithm well, it is useful to understand it from several distinct points of view. One of them is the static point of view on which correctness proofs are based: Formulate invariants on the data and show that these are preserved under the program's operations. This abstract approach appeals to our rational mind. A second, equally important point of view, is dynamic: Watch the algorithm go through its paces on a variety of input data. This concrete approach appeals to our intuition. Whereas the static approach relies mainly on "thinking", the dynamic approach calls mostly for "doing" and "perceiving", and thus is a prime candidate for visual human-

computer interaction. In this use of algorithm animation, the user may be checking his understanding of the algorithm, or may be checking the algorithm's correctness—in principle, he could reason this out, but in practice, it is faster and safer to have the computer animation as a double check.

Exploring

In a growing number of applications, computer visualization cannot be replaced by any other technique. This is the case, for example, in exploratory data analysis, where a scientist may not know a priori what she is looking for, and the only way to look at a mass of data is to generate pictures from it (see a special issue on scientific visualization [Nie 89]). At times static pictures will do, but in simulations (e.g. of the onset of turbulent flow) we prefer to see an animation over time.

Turning to the *techniques of animation*, computer technology is in the midst of extremely rapid evolution toward ever-higher-quality interactive image generation on powerful graphics workstations (see [RN 91] for a survey of the state of the art). Fortunately, animating algorithms such as those presented in this book can be done adequately with the graphics tools available on low-cost workstations. These algorithms operate on discrete data configurations (such as matrices, trees, graphs), and use standard data structures, such as arrays and lists. For such limited classes of algorithms, there are software packages that help produce animations based on specifications, with a minimum of extra programming required. An example of an algorithm animation environment is the BALSA system [Bro 88, BS 85]. A more recent example is the XYZ GeoBench, which animates geometric algorithms [NSDAB 91].

In our experience, the bottleneck of algorithm animation is not the extra code required, but graphic design. *What do you want to show, and how do you display it, keeping in mind the limitations of the system you have to work with?* The key point to consider is that data does not look like anything until we have defined a mapping from the data space into visual space. Defining such a mapping ranges from trivial to practically impossible.

1. For some kinds of data, such as geometric data in two- and three-dimensional space, or real-valued functions of one or two real variables, there are natural mappings that we learned in school. These help us greatly in getting a feel for the data.
2. Multidimensional data (dimension ≥ 3) can be displayed on a two-dimensional screen using a number of straight forward techniques, such as projections into a subspace, or using color or gray level as a fourth dimension. But our power of perception diminishes rapidly with increasing dimensionality.
3. For discrete combinatorial data there is often no natural or accepted visual representation. As an example, we often draw a graph by mapping nodes into points and edges into lines. This representation is natural for graphs that are embedded in Euclidean space, such as a road network, and we can readily make sense of a map with thousands of cities and road links. When we extend it to arbitrary graphs by placing a node anywhere on the screen, on the other hand, we get a random crisscrossing of lines of little intuitive value.

In addition to such inherent problems of visual representation, practical difficulties of the most varied type abound. *Examples:*

- Some screens are awfully small, and some data sets are awfully large for display even on the largest screens.
- An animation has to run within a narrow speed range. If it is too fast, we fail to follow, or the screen may flicker disturbingly; if too slow, we may lack the time to observe it.

In conclusion, we hold that it is not too difficult to animate simple algorithms as discussed here by interspersing drawing statements into the normal code. Independent of the algorithm to be animated, you can call on your own collection of display and interaction procedures that you have built up in your frame program (in the section "A graphics frame program). But designing an adequate graphic representation is hard and requires a creative effort for each algorithm—that is where animators/programmers will spend the bulk of their effort. More on this topic in [NVH 86].

Example: the convex hull of points in the plane

The following program is an illustrative example for algorithm animation. 'ConvexHull' animates an *on-line* algorithm that constructs half the convex hull (say, the upper half) of a set of points presented incrementally. It accepts one point at a time, which must lie to the right of all preceding ones, and immediately extends the convex hull. The algorithm is explained in detail in “sample problems and algorithms”.

```
program ConvexHull;  { of n ≤ 20 points in two dimensions }

const  nmax = 19;  { max number of points }
       r = 3; { radius of point plot }
var  x, y, dx, dy: array[0 .. nmax] of integer;
       b: array[0 .. nmax] of integer;  { backpointer }
       n: integer;  { number of points entered so far }
       px, py: integer;  { new point }

procedure PointZero;
begin
  n := 0;
  x[0] := 5;  y[0] := 20;  { the first point at fixed location }
  dx[0] := 0;  dy[0] := 1;  { assume vertical tangent }
  b[0] := 0;  { points back to itself }
  PaintOval(y[0] - r, x[0] - r, y[0] + r, x[0] + r)
end;

function NextRight: boolean;
begin
  if  n ≥ nmax  then
    NextRight := false
  else  begin
    repeat until Button;
    while Button do  GetMouse(px, py);
    if  px ≤ x[n]  then
      NextRight := false
    else  begin
      PaintOval(py - r, px - r, py + r, px + r);
      n := n + 1;  x[n] := px;  y[n] := py;
      dx[n] := x[n] - x[n - 1]; { dx > 0 } dy[n] := y[n] - y[n -1];
      b[n] := n - 1;
      MoveTo(px, py);  Line(-dx[n], -dy[n]);  NextRight := true
    end
  end
end;

procedure ComputeTangent;
var i: integer;
begin
  i := b[n];
  while  dy[n] · dx[i] > dy[i] · dx[n]  do  begin  { dy[n]/dx[n] >
dy[i]/dx[i] }
```

```
      i := b[i];
      dx[n] := x[n] - x[i];  dy[n] := y[n] - y[i];
      MoveTo(px, py);  Line(-dx[n], -dy[n]);
      b[n] := i
    end;
    MoveTo(px, py);  PenSize(2, 2);  Line(-dx[n], -dy[n]);  PenNormal
end;

procedure Title;
begin
    ShowText;  ShowDrawing;  { make sure windows lie on top }
    WriteLn('The convex hull');
    WriteLn('of n points in the plane sorted by x-coordinate');
    WriteLn('is computed in linear time.');
    Write('Click next point to the right, or Click left to quit.')
end;

begin  { ConvexHull }
    Title;  PointZero;
    while  NextRight  do  ComputeTangent;
    Write('That's it!')
    end.
```

A gallery of algorithm snapshots

The screen dumps shown in Exhibit 3.1 were taken from demonstration programs that we use to illustrate topics discussed in class. Although snapshots cannot convey the information and the impact of animations, they may give the reader ideas to try out. We select two standard algorithm animation topics (sorting and random number generation), and an example showing the effect of cumulative rounding errors.

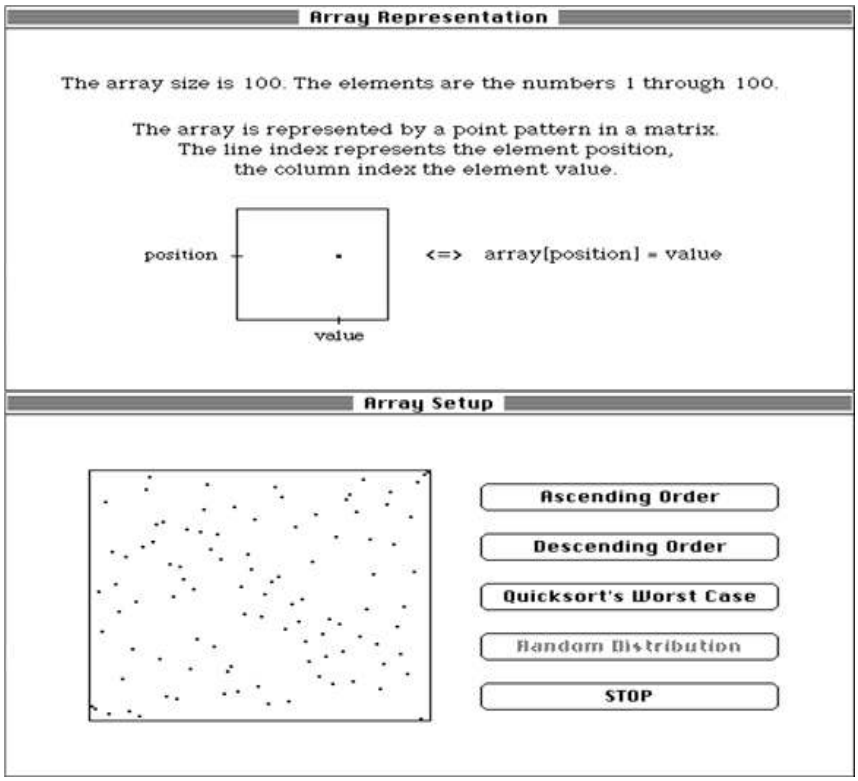


Exhibit 3.1: Initial configuration of data, ...

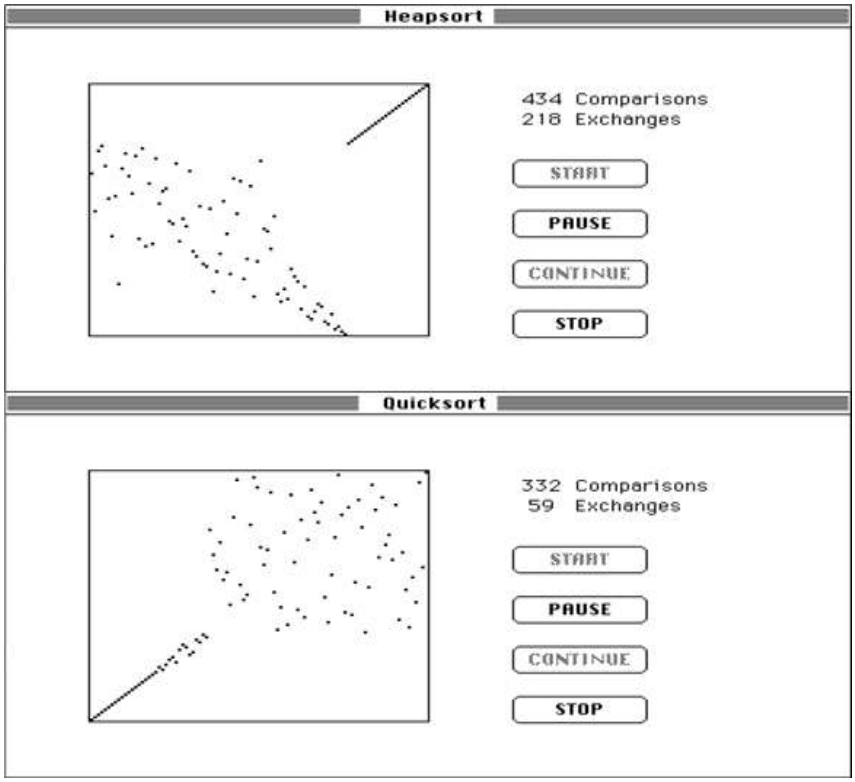


Exhibit 3.1: ... and snapshots from two sorting algorithms.

Visual test for randomness

Our visual system is amazingly powerful at detecting patterns of certain kinds in the midst of noise. Random number generators (RNGs) are intended to simulate "noise" by means of simple formulas. When patterns appear in the visual representation of supposedly random numbers, chances are that this RNG will also fail more rigorous statistical tests. The eyes' pattern detection ability serves well to disqualify a faulty RNG but cannot certify one as adequate. Exhibit 3.2 shows a simulation of the Galton board. In theory, the resulting density diagram should approximate a bellshaped Gaussian distribution. Obviously, the RNG used falls short of expectations.

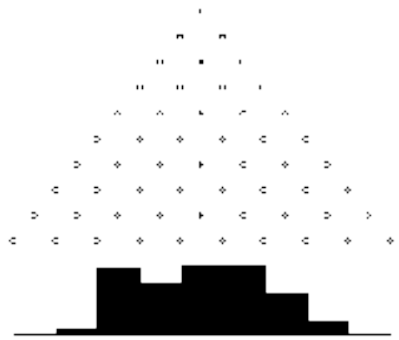


Exhibit 3.2: One look suffices to unmask a bad RNG.

Numerics of chaos, or chaos of numerical computation?

The following example shows the effect of rounding errors and precision in linear recurrence relations. The d-step linear recurrence with constant coefficients in the domain of real or complex numbers,

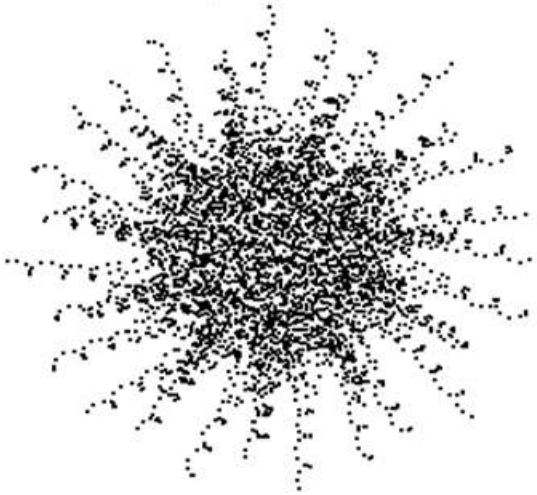
$$z_k = \sum_{i=1}^{d} c_i z_{k-i}, \quad c_i \in C$$

is one of the most frequent formulas evaluated in scientific and technical computation (e.g. for the solution of differential equations). By proper choice of the constants c_i and of initial values $z_0, z_1, \ldots, z_{d-1}$ we can generate sequences z_k that when plotted in the plane of complex numbers form many different figures. With d= 1 and $|\chi_1| = 1$, for example, we generate circles. The pictures in Exhibit 3.3 were all generated with d = 3 and conditions that determine a curve that is most easily described as a circle 3 running around the perimeter of another circle 2 that runs around a stationary circle 1. We performed this computation with a floating-point package that lets us pick precision P (i.e. the number of bits in the mantissa). The resulting pictures look a bit chaotic, with a behavior we have come to associate with fractals—even if the mathematics of generating them is completely different, and linear recurrences computed without error would look much more regular. Notice that the first two images are generated by the same formula, with a single bit of difference in the precision used. The whim of this 1-bit difference in precision changes the image entirely.

B=2, P=12
N1=3, R1=15
N2=5, R2=40
N3=7, R3=10
Iterations: 1993

B=2, P=13
N1=3, R1=15
N2=5, R2=40
N3=7, R3=10
Iterations: 14498

B=2, P=15
N1=5, R1=10
N2=7, R2=50
N3=11, R3=20
Iterations: 4357



3. Algorithm animation

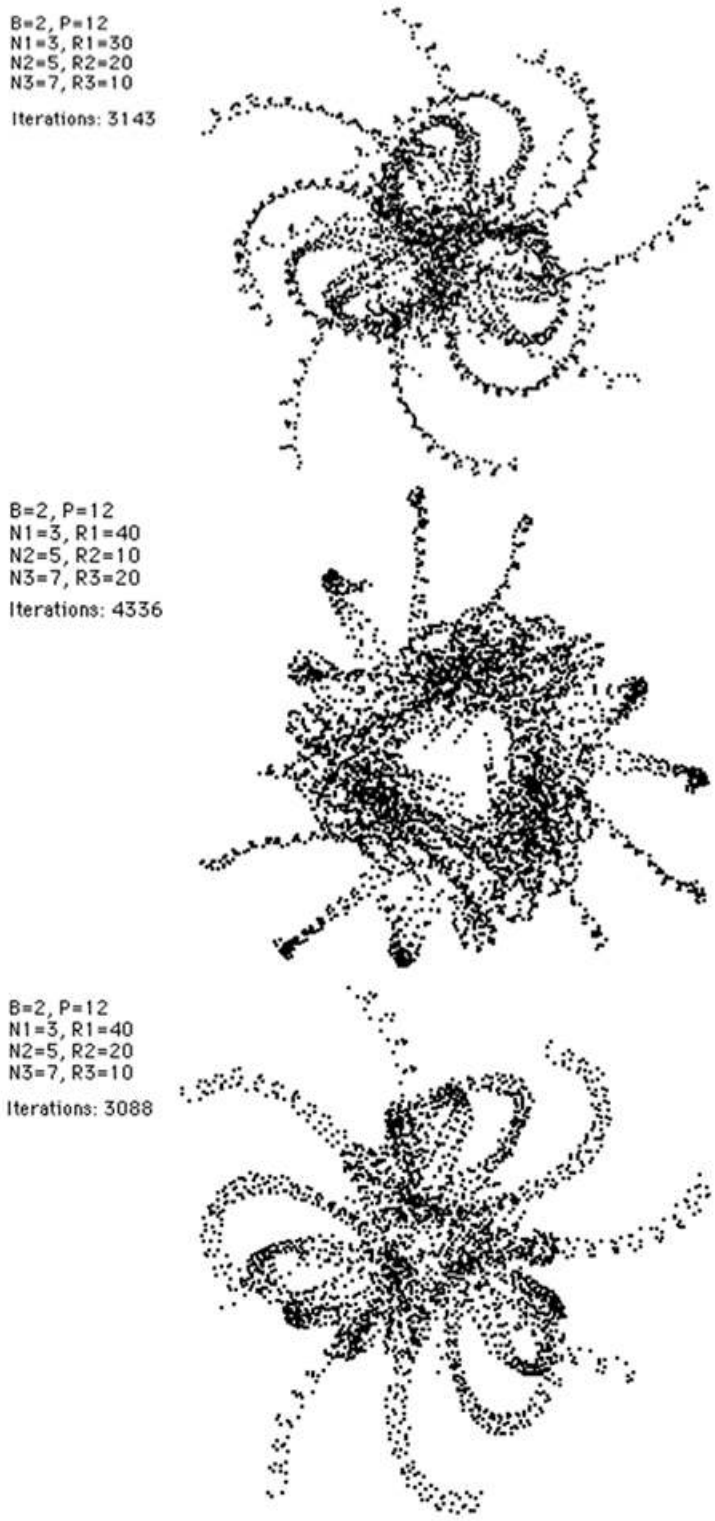


Exhibit 3.3: The effect of rounding errors in linear recurrence relations.

Programming projects

1. Use your personal graphics frame program (the programming project of "graphics primitives and environments") to implement and animate the convex hull algorithm example.

2. Use your graphics frame program to implement and animate the behavior of recurrence relations as discussed in the section “A gallery of algorithm snapshots”.
3. Extend your graphics frame program with a set of dialog control operations sufficient to guide the user through the various steps of the animation of recurrence relations: in particular, to give him the options, at any time, to enter a new set of parameters, then execute the algorithm and animate it in either 'movie mode' (it runs at a predetermined speed until stopped by the user), or 'step mode' [the display changes only when the user enters a logical command 'next' (e.g. by clicking the mouse or hitting a specific key)].