

# Algorithms

J. H. WEGSTEIN, Editor

## ALGORITHM 84 SIMPSON'S INTEGRATION

PAUL E. HENNION

Giannini Controls Corporation  
Astromechanics Research Division, Berwyn, Penn.

**real procedure** SIM (n, a, b, y);  
**value** n, a, b; **real** a, b; **integer** n; **array** y;  
**comment** This is a method for obtaining the approximate value of the definite integral of a continuous function when the integral cannot be evaluated in elementary functions. Given  $y = f(x)$  and the  $\int_a^b y \, dx$  to be evaluated. Plot the curve  $f(x)$ , and divide  $[a, b]$  evenly into  $n$  equal parts, erecting the ordinates  $y_0, y_1, \dots, y_n$ . Then the approximate value of the definite integral by Simpson's rule states that:

$$\int_a^b f(x) \, dx = \frac{b-a}{3n} (y_0 + 4y_1 + 2y_2 + \dots + 4y_{n-1} + y_n);$$

**begin** **real** s; **integer** i;  
s := (y[0] - y[n])/2;  
**for** i := 1 **step** 2 **until** n - 1 **do** s := s + 2 × y[i] + y[i+1];  
SIM := 2 × (b - a) × s/(3 × n)  
**end**

## ALGORITHM 85 JACOBI

THOMAS G. EVANS

Bolt, Beranek, and Newman\*, Cambridge, Mass.

\* This work has been sponsored by the Air Force Cambridge Research Laboratories, OAR (USAF), Detection Physics Laboratory, under contract AF 19(628)-227.

**procedure** JACOBI (A, S, n, rho);  
**value** n, rho; **integer** n; **real** rho; **real array** A, S;  
**comment** This procedure finds all eigenvalues and eigenvectors of a given square symmetric matrix by a modified Jacobi (iterative) method (cf. J. Greenstadt, "The determination of the characteristic roots of a matrix by the Jacobi method," in *Mathematical Methods for Digital Computers*, A. Ralston and H. S. Wilf, eds.). JACOBI is given a square symmetric matrix of order  $n$  stored in the array A. The initial contents of the array S are immaterial, as S is initialized by the procedure. At exit the  $k^{\text{th}}$  column of the array S contains the  $k^{\text{th}}$  of the  $n$  eigenvectors of the given matrix, and the diagonal element  $A[k, k]$  of the array A is the corresponding  $k^{\text{th}}$  eigenvalue. The parameter rho is the "accuracy requirement" introduced in the above reference, where a detailed flow chart of the method is given. The significance of rho is that the iteration terminates when, for every off-diagonal element  $A[i, j]$ ,  $\text{abs}(A[i, j]) < (\text{rho}/n) \times \text{norm1}$ , where norm1 is a function only of the off-diagonal elements of the original matrix;  
**begin** **real** norm1, norm2, thr, mu, omega, sint, cost, int1, v1, v2, v3;  
**integer** i, j, p, q, ind;  
**comment** Set array S =  $n \times n$  identity matrix;  
**for** i := 1 **step** 1 **until** n **do**

**for** j := 1 **step** 1 **until** i **do**  
  **if** i = j **then** S[i, j] := 1.0  
  **else** S[i, j] := S[j, i] := 0.0;  
**comment** Calculate initial norm (norm1), final norm (norm2), and threshold (thr);  
int1 := 0.0;  
**for** i := 2 **step** 1 **until** n **do**  
  **for** j := **step** 1 **until** i-1 **do**  
    int1 := int1 + 2.0 × A[i, j] ↑ 2;  
  norm1 := sqrt(int1); norm2 := (rho/n) × norm1;  
  thr := norm1; ind := 0;  
main: thr := thr/n;  
  **comment** The sweep through the off-diagonal elements begins here;  
main1: **for** q := 2 **step** 1 **until** n **do**  
  **for** p := 1 **step** 1 **until** q-1 **do**  
    **if** abs(A[p, q]) ≥ thr **then**  
      **begin** ind := 1; v1 := A[p, p]; v2 := A[p, q];  
      v3 := A[q, q]; mu := 0.5 × (v1 - v3);  
      omega := (if mu = 0.0 **then** 1 **else** sign(mu)) ×  
      (-v2)/sqrt(v2↑2 + mu↑2);  
      sint := omega/sqrt(2.0 × (1.0 + sqrt(1.0 -  
      omega↑2)));  
      cost := sqrt(1.0 - sint↑2);  
      **for** i := 1 **step** 1 **until** n **do**  
        **begin** int1 := A[i, p] × cost - A[i, q] × sint;  
          A[i, q] := A[i, p] × sint + A[i, q] × cost;  
          A[i, p] := int1;  
          int1 := S[i, p] × cost - S[i, q] × sint;  
          S[i, q] := S[i, p] × sint + S[i, q] × cost;  
          S[i, p] := int1  
        **end**;  
      **for** i := **step** 1 **until** n **do**  
        **begin** A[p, i] := A[i, p]; A[q, i] := A[i, q] **end**;  
        A[p, p] := v1 × cost↑2 + v3 × sint↑2 - 2.0 ×  
        v2 × sint × cost;  
        A[q, q] := v1 × sint↑2 + v3 × cost↑2 + 2.0 ×  
        v2 × sint × cost;  
        A[p, q] := A[q, p] := (v1 - v3) × sint × cost +  
        v2 × (cost↑2 - sint↑2)  
      **end**;  
      **comment** Now test to see if current tolerance exceeded and,  
      if not, whether final tolerance reached;  
      **if** ind = 1 **then** **begin** ind := 0; **go to** main1 **end**  
      **else if** thr > norm2 **then** **go to** main  
    **end** JACOBI

## ALGORITHM 86 PERMUTE

J. E. L. PECK AND G. F. SCHRACK

University of Alberta, Calgary, Alberta, Canada

**procedure** PERMUTE (x, n);  
**array** x; **integer** n;  
**comment** Each call of PERMUTE executes a permutation of the first  $n$  components of  $x$ . It assumes a nonlocal Boolean variable 'first', which when true causes the procedure to initialise the signature vector p. Thereafter 'first' remains false until after  $n!$  calls;  
**begin** **own integer array** p[2:n]; **integer** i, k;  
  **if** first **then**  
    **begin** **for** i := 2 **step** 1 **until** n **do**  
      p[i] := i; first := false  
    **end** initialise;  
  **for** k := 2 **step** 1 **until** n **do**

```

begin integer km; real t;
  t := x[1]; km := k - 1;
  for i := 1 step 1 until km do
    x[i] := x[i+1];
  x[k] := t; p[k] := p[k] - 1;
  if p[k] ≠ 0 then go to EXIT;
  p[k] := k
end k;
first := true;
EXIT: end PERMUTE

```

## ALGORITHM 87 PERMUTATION GENERATOR

JOHN R. HOWELL

Orlando Aerospace Division, Martin Marietta Corp.,  
Orlando, Florida

**procedure** PERMUTATION (N, K);

**value** K, N; **integer** K; **integer array** N;

**comment** This **procedure** generates the next permutation in lexicographic order from a given permutation of the K marks 0, 1, ..., (K-1) by the repeated addition of (K-1) radix K. The radix K arithmetic is simulated by the addition of 9 radix 10 and a test to determine if the sum consists of only the original K digits. Before each entry into the **procedure** the K marks are assumed to have been previously specified either by input data or as the result of a previous entry. Upon each such entry a new permutation is stored in N[1] through N[K]. In case the given permutation is (K-1), (K-2), ..., 1, 0, then the next permutation is taken to be 0, 1, ..., (K-1). A FORTRAN subroutine for the IBM 7090 has been written and tested for several examples;

**begin integer** i, j, carry;

**for** i := 1 **step** 1 **until** K **do**

**if** N[i] - K + i ≠ 0 **then go to** add;

**for** i := 1 **step** 1 **until** K **do** N[i] := i - 1;

**go to** exit;

add: N[K] := N[K] + 9;

**for** i := 1 **step** 1 **until** K-1 **do**

**begin if** K > 10 **then go to** B;

      carry := N[K-i+1] ÷ 10; **go to** C;

    B: carry := N[K-i+1] ÷ K;

    C: **if** carry = 0 **then go to** test;

      N[K-i] := N[K-i] + carry;

      N[K-i+1] := N[K-i+1] - 10 × carry

**end if**;

test: **for** i := 1 **step** 1 **until** K **do if** N[i] - (K - 1) > 0 **then go to** add;

**for** i := 1 **step** 1 **until** K-1 **do**

**for** j := i+1 **step** 1 **until** K **do**

**if** N[i] - N[j] = 0 **then go to** add;

exit: **end** PERMUTATION GENERATOR

## CERTIFICATION OF ALGORITHM 35

SIEVE (T. C. Wood, *Comm. ACM*, March 1961)

P. J. BROWN

University of North Carolina, Chapel Hill, N. C.

SIEVE was transliterated into GAT for the UNIVAC 1105 and successfully run for a number of cases.

The statement:

**go to if** n/p[i] = n ÷ p[i] **then** b1 **else** b2;

was changed to the statement:

**go to if** n/p[i] - n ÷ p[i] < .5/N<sub>max</sub> **then** b1 **else** b2;

Roundoff error might lead to the former giving undesired results.

## CERTIFICATION OF ALGORITHM 71

PERMUTATION (R. R. Coveyou and J. G. Sullivan,

*Comm. ACM*, Nov. 1961)

P. J. BROWN

University of North Carolina, Chapel Hill, N. C.

PERMUTATION was transliterated into GAT for the UNIVAC 1105 and successfully run for a number of cases.

## CERTIFICATION OF ALGORITHM 71

PERMUTATION (R. R. Coveyou and J. G. Sullivan,

*Comm. ACM*, Nov. 1961)

J. E. L. PECK AND G. F. SCHRACK

University of Alberta, Calgary, Alberta, Canada

PERMUTATION was translated into FORTRAN for the IBM 1620 and it performed satisfactorily. The **own integer array** x[0:n] may be shortened to x[1:n], provided corresponding corrections are made in the first two **for** statements.

However, PERMUTE (Algorithm 86) is superior to PERMUTATION in two respects.

(1) PERMUTATION, using storage of order 2n, is designed to permute the specific vector 0, 1, 2, ..., n-1 rather than an arbitrary vector. Thus storage of order 3n is required to permute an arbitrary vector. PERMUTE, in contrast, only needs storage of order 2n to permute an arbitrary vector.

(2) PERMUTE is built up from cyclic permutations. The number of permutations actually executed internally (the redundant ones are suppressed) by PERMUTE is asymptotic to (e-1)n! rather than n!. In spite of this, PERMUTE is distinctly faster (1316 against 2823 seconds for n = 8) than PERMUTATION. If t<sub>n</sub> is the time taken for all permutations of a vector with n components, and if r<sub>n</sub> = t<sub>n</sub>/nt<sub>n-1</sub>, then one would expect r<sub>n</sub> to be close to 1. Experiment with small values of n gave the following results for r<sub>n</sub>.

	n	6	7	8
PERMUTE		0.96	0.99	1.00
PERMUTATION		1.10	1.13	1.12

Is there yet a faster way to do it?

See also: C. Tompkins, "Machine Attacks on Problems whose Variables are Permutations", *Proceedings of Symposia in Applied Mathematics*, Vol. VI: *Numerical Analysis* (N. Y., McGraw-Hill, 1956).

## The Calculation of Easter...

By Donald Knuth

California Institute of Technology, Pasadena, California

Here are two programs, written to demonstrate ALGOL and COBOL. Object: to determine the month and day of Easter, given the year. The ALGOL program (1) is written as a procedure, which sets up "month" and "day" given the value of "year." The COBOL program (2) prepares a printed table of Easter date, from 500 to 4999 A.D.

(1)

**procedure** Easter (year, month, day); **value** year; **integer** year, month, day;

**comment** This procedure calculates the day and month of Easter given the year. It gives the actual date of "Western Easter" (not the Eastern Easter of the Eastern Orthodox churches) after A.D. 463. "golden number" is the number of the year in the Metonic cycle, used to determine the position of the calendar moon. "Gregorian correction" is the number of preceding years like 1700, 1800, 1900 when leap year was not held, "Clavian correction" is a correction for the Metonic cycle of about