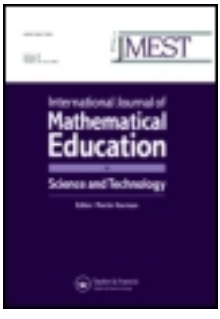


International Journal of Mathematical Education in Science and Technology
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tmes20

# Specifying and simulating rotor machines using Maple

G. P. McKeown

Available online: 11 Nov 2010

To cite this article: G. P. McKeown (1999): Specifying and simulating rotor machines using Maple, International Journal of Mathematical Education in Science and Technology, 30:6, 867-887

To link to this article: http://dx.doi.org/10.1080/002073999287545

INT. J. MATH. EDUC. SCI. TECHNOL., 1999, VOL. 30, NO. 6, 867–887

# Specifying and simulating rotor machines using Maple

G. P. MCKEOWN
School of Information Systems, University East Anglia, Norwich, NR4 7TJ, England
Email: gpm@sys.uea.ac.uk

(*Received 17 August 1998; Revised 6 January 1999*)

The symbolic computation language, Maple, is used to specify and simulate the performance of time-varying substitution cipher systems known as rotor machines, the most famous examples of which are the ENIGMA machines used by Germany during World War II. A mathematical model is presented, initially for a one-rotor machine. The direct implementation of this model in Maple provides a prototype system from which more efficient implementations are derived via a process of program transformation. The one-rotor system is then generalized to an $N$-rotor system. The paper provides a tutorial on the application of symbolic computation to a problem of interest to students of mathematics and of computer science.



## 1. Introduction

Mathematical computation has two components: numerical computation and symbolic computation. The former involves the execution of arithmetic operations on numbers and has been a fundamental activity both in prompting the development of computers and in their subsequent application since the beginning of the computer age. As its name implies, symbolic computation involves the use of symbols to represent mathematical entities, followed by the algebraic manipulation of those symbols. The use of computers for symbolic computation is a much more recent phenomenon than their use for numerical computation. The earliest general-purpose symbolic computation systems date from the late 1960s [1, 2]. The 1980s saw the development of the first versions of systems such as *Maple* [3, 4] and *Mathematica* (see, for example, [5]) which integrate symbolic, numeric and graphic computations in a way designed to provide users with easy-to-use environments for doing mathematical computation. Here, we shall be concerned only with the Maple V system (Release 4). A good review of other current symbolic computation systems is given in [6]. Because the Maple system also includes a procedural language, it provides an excellent medium for developing rapid prototypes of software systems and for specifying and simulating special-purpose hardware systems. The purpose of this paper is to illustrate the latter using a case study which should appeal to students. As we present the mathematics underlying our case study device, so we implement the mathematical description directly in Maple. Although such a direct approach does not lead to a particularly efficient Maple implementation, we are then able to illustrate the important idea of program transformation by systematically transforming our initial implementation to obtain a more efficient Maple program.

*International Journal of Mathematical Education in Science and Technology*
ISSN 0020–739X print/ISSN 1464–5211 online © 1999 Taylor & Francis Ltd
http://www.tandf.co.uk/JNLS/mes.htm
http://www.taylorandfrancis.com/JNLS/mes.htm

The variety of uses to which computers are put is constantly increasing. One of the earliest uses, however, and the one which during World War II provided the final impetus that soon after led to the development of the first general-purpose computers, is cryptography, or more precisely *cryptology*, which comprises cryptography and cryptanalysis. *Cryptography* is the process of using encipherment to keep messages secure. The word itself comes from the Greek for 'hidden writing' and as its ancient name implies, cryptography has been used for thousands of years. As long ago as 360 BC, a Roman called Aeneas Tacticus wrote a military manual on the art of war which included a chapter entitled 'On Secret Messages'. Messages to be enciphered are referred to as *plaintext* while encrypted messages are called *ciphertext*. *Cryptanalysis* is the art and science of breaking ciphertext. The simplest type of cipher is a substitution cipher in which each letter in the plaintext alphabet is mapped to a different letter. More secure ciphers are derived by using different simple substitution ciphers to encrypt different parts of a plaintext message. A rotor machine is an electromechanical device for implementing such time-varying substitution ciphers. The ENIGMA machines used by Germany during World War II were rotor machines. An excellent discussion of the ENIGMA machine and of the poor operational practice that made it insecure is given by Churchhouse [7, 8]. The cryptanalysis of rotor machines is also discussed in [9, 10].



## 2. Maple

Maple is a symbolic computation system that can be run on a variety of computer platforms ranging from PCs and Macs to supercomputers. The system comprises four components: a user interface, known as the *Iris*; a core algebraic engine, known as the *kernel*; an external library, and a share library contributed by users. The Iris and the kernel are written in C and are loaded at the start of every Maple session. The external library contains an extensive number of mathematical functions, many of them grouped in appropriate packages. For example, the *numtheory* package collects useful functions for performing computations in number theory, while the *linalg* package is a collection of functions for performing linear algebra operations. Packages and individual functions from the external library must be loaded explicitly if they are required during a Maple session. The functions in the external library are written in Maple's own Pascal-like language. This language contains the usual constructs of a procedural language, such as *if* statements, *for* statements and *while* statements. Users can use this language to create their own commands and by collecting the definitions of a set of such commands in a file, a user-defined package is created. In this paper we describe a Maple package that we have created for specifying and simulating rotor machines.

It is not the intention of this paper to provide a tutorial on Maple, and we discuss only those Maple commands relevant to our discussion of rotor machines. For a comprehensive introduction to Maple the reader is referred to [11]. Before beginning our Maple-based presentation of rotor machines, however, we do make a few general points about using Maple and introduce a few useful general commands. When a Maple session is started, for example by typing *maple* or *xmaple* from a shell on a Unix machine, the system responds with a prompt, such as the symbol >. Each command typed in response to such a prompt must be terminated by either a semi-colon (;) or a colon (:). The former method of

termination causes the result of the command to be displayed while a colon terminator suppresses this output to the screen. In this paper, we use a typewriter font to distinguish commands typed by the user. Any output that would be displayed by the Maple system in response to such a command is given immediately after the command, indented and in normal type. One way of including a comment in a Maple session is to type a sharp symbol, #, followed by the required comment.

Matrices play an important part in the mathematical model for rotor machines presented in the next section, so we mention here the special evaluation rules relating to arrays in Maple. If the name, $A$, has been assigned an array then $A$ evaluates to the name, $A$. To obtain the actual array structure, we can use the *eval* command:

```
>A := array(1..3, 1..3, [[1, 2, 3], [4, 5, 6], [7, 8, 9]]);
```

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
>A; # Last name evaluation of A
```

$$A$$

```
>eval (A); # Fully evaluate A
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

The Maple *op* command is used to extract operands from an expression, while the *nops* command gives the number of operands in a given expression. For example, if *nops* is applied to a list, it delivers the number of elements in the list, i.e. the length of the list:

```
> ls := [1, 2, 4, 8, 16]; # Assign a list of integers to ls
```

$$ls := [1, 2, 4, 8, 16]$$

```
>nops(ls); # Determine the length of ls
```

$$5$$

As is the case for many Maple functions, *op* is *polymorphic*, i.e. has many forms of usage. Two of the possible forms of use of *op* are *op(e)* and *op(i,e)*, where *e* is an expression and *i* is a non-negative integer. The result of *op(e)* is a sequence of all of the operands of *e*. *op(i,e)* delivers the *i*th operand of *e*, unless $i = 0$, in which case the result is the type of *e*.

```
>op(ls); # The operands of a list are its elements.
```

$$1, 2, 4, 8, 16$$

```
>op(3, ls);
```

$$4$$

```
>sum(op(i, ls), i=1..nops(ls)); # Sum the elements of ls
```

31

```
> ls := [op(ls), 32]; # Append 32 to list ls
```

$$ls := [1, 2, 4, 8, 16, 32]$$

```
>op(0, ls); # Obtain the type of ls
```

*list*

Lists play an important part in many of the functions defined in this paper. One of the operations on lists which we will have occasion to use is that of reversing a list. If *ls* is a list of length *n* then the reversal of *ls*, *rv_ls*, may be computed in Maple as follows:

```
> rv_ls := [seq(ls[-i]), i=1..n]:
```

*ls[−1]* gives the last element of *ls*, *ls[−2]* gives the second from last element of *ls*, etc. Hence, *seq(ls[-i]), i = 1..n* generates the sequence consisting of the elements of *ls* in reverse order. Applying square brackets to this sequence converts it into a list.

A user may create his or her own command by encapsulating the required operation within a procedure written in the Maple programming language. A Maple procedure has the following general form:

*op_name* `proc` ( *param_seq* )
  [*local loc_name_seq;*]
  [*global glob_name-seq;*]
  [*options opt-name-seq;*]
  statements
*end*

Here, *op_name* is a suitable name for the new command. *param_seq* may be a sequence of Maple identifiers separated by commas. Alternatively, each identifier in *param_seq* may be qualified by its type, as illustrated in the following example:

```
k::integer, R::matrix
```

There are three types of declaration in a procedure, each of which is optional and consists of a keyword (*local*, *global* or *options*) followed by a comma-separated list of Maple identifiers. *local* declares all variables considered local to the procedure, while *global* declares all variables considered global. The *options* declaration is included if the user wishes to specify one or more of a number of options. For example, if the *remember* option is specified, then every time the procedure is called with a new sequence of actual parameters, an entry is placed in a remember table. Storing previously computed results can greatly increase the execution speed of subsequent calls to the procedure, particularly in the case of recursive procedures. Every built-in Maple command has an associated remember table and this option is also included in all of the procedures defined in this paper. If a *RETURN* command is encountered within *statements*, execution of the procedure terminates: if the *RETURN* command has an argument, then the value returned is the value of that argument; otherwise, the value returned is the last value computed before the *RETURN* command. In the absence of a *RETURN* command, the value returned by the procedure is the result of the execution of the last command in *statements*.

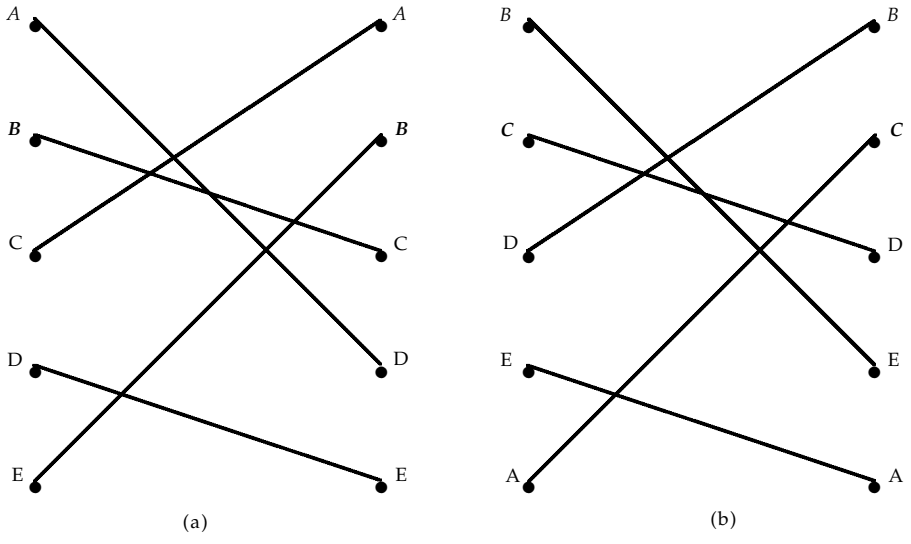


Figure 1. (a) Before and (b) after a cyclic shift through one position.



## 3. Maple Simulation of Rotor Machines

A *rotor machine* is an electromechanical device through which a message is encrypted using a substitution cipher that varies as each letter is enciphered. This time-varying substitution cipher system is achieved by means of a bank of rotors rotating on a common shaft. Each rotor is a disc with $q$ evenly spaced electrical contacts around the perimeter of each face, where $q$ is the size of the plaintext alphabet, $\mathcal{A}$. Each contact on the front face of a rotor is internally wired to exactly one contact on the rear face. An electrical signal corresponding to a letter of the plaintext alphabet enters at one end of the bank of rotors and passes through each of the rotors in turn. In effect, letters are permuted as they pass through rotors. If all of the rotors were fixed, the effect of the bank of rotors would be only that of a simple substitution cipher. However, rotating any rotor to a new position changes the underlying permutation, and so changes the substitution cipher (see figure 1 for a simple one-rotor example in which $q = 5$). In a rotor machine, at least one of the rotors is rotated through one or more positions each time a plaintext letter is enciphered (or deciphered). We will assume that for any rotor, one contact point is designated as a base point. We will then say that the rotor is in its *base position* when the first letter of the plaintext alphabet corresponds to the base point. Thus, if the rotor in figure 1(a) is in its base position, then in figure 1(b), the rotor is cyclically shifted by one position from its base position.

### 3.1. *Mathematical model using Maple*

We begin by developing a model for a one-rotor machine. We will make use of Maple commands that we have defined and collected in a package called *rotor_pkg*. Many of the procedures in this package make use of commands from Maple's *linalg* package, so we begin by loading this package:

```
> with(linalg): # load Maple's linear algebra package
```

Note that as an alternative to loading the entire *linalg* package, we could use the long form of the name for each of the *linalg* commands used within *rotor_pkg* functions. For example, if the *linalg* package has been loaded, we can invert a matrix, $A$, as follows:

```
> inverse(A);
```

If the *linalg* package has not been loaded, we could invert $A$ as follows:

```
> linalg[inverse] (A);
```

For the remainder of this paper, we assume that the *linalg* package has been loaded as above. Next we load our rotor package. Since this is a user-defined package, it is loaded using Maple's *read* command:

```
> read(rotor_pkg): # load the rotor package
```

As a simple illustration, we define an alphabet of size five comprising the first 5 lower-case letters of the English alphabet:

```
> sz := 5:
> alphabet := [a, b, c, d, e]:
```

More generally, we might take an alphabet comprising all 26 lower-case letters. Consequently, during our simulated Maple session, we will not use any identifier consisting of a single lower-case letter for any other purpose. Without loss of generality, we may associate an encoding of 1 with *a*, 2 with *b*, etc. We may do this in Maple as follows:

```
> for cd from 1 to sz do encoding[alphabet[cd]] := cd od;
```

$$encoding_a := 2$$
$$encoding_b := 2$$
$$encoding_c := 3$$
$$encoding_d := 4$$
$$encoding_e := 5$$

The fixed rotor connections may then be represented by a permutation of 1 to 5 (for the example in figure 1, the permutation is $[4, 3, 1, 5, 2]$). Such a permutation may in turn be represented by a permutation matrix. This is effected by the *rotor_pkg* command *mk_rotor* defined in figure 2. For the example illustrated in figure 1:

```
> R := ml_rotor(sz, [4, 3, 1, 5, 2]): eval(R);
```

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

We refer to $R$ as the *rotor permutation matrix*. Note the use of the colon (:) terminator after the *mk_rotor* command to suppress the last name evaluation of $R$, followed by the use of the *eval* command to obtain a full evaluation of $R$. The

```
# Given the underlying permutation, p, of size
# q, representing the fixed rotor connections,
# mk_rotor generates the corresponding rotor
# permutation matrix.
mk_rotor := proc(q::integer, p::list)
  local R, i;
  options remember;
  R := matrix(q,q,0); # Initialize R to the q by q
                      # zero matrix.
  for i from 1 to q do R[p[i],i]:= 1 od;
  RETURN(R)
end;
```

Figure 2. The *rotor_pkg* function *mk_rotor*.



```
# Given the alphabet size, q, mk_cyclic_mat
# delivers a cyclic matrix of that size.
mk_cyclic_mat := proc(q::integer)
  local C, i;
  options remember;
  C:= matrix(q,q,0);
  C[1,q] := 1;
  for i from 2 to q do
    C[i,i-1] := 1
  od;
  RETURN(C)
end;
```

Figure 3. The *rotor_pkg* function *mk_cyclic_mat*.

restriction mentioned above on the use of names consisting of a single lower-case letter does not apply to formal arguments in a procedure, nor to local variables in a procedure.

When the rotor is in its base position, $R$ also represents the current *enciphering permutation matrix*, i.e. the matrix form of the underlying permutation of the current substitution cipher. If the rotor is then rotated through one position, the enciphering permutation matrix (and hence the substitution cipher) is changed. This change in the underlying permutation corresponds to a similarity transformation of $R$:

$$S(1, R) = CRC^{-1}$$

Here, $C$ is a permutation matrix obtained by cyclically left-shifting the columns of the $sz \times sz$ identity matrix through one position. The latter may be achieved by the *rotor_pkg* command *mk_cyclic_mat* (see figure 3). For our example:

```
> C := mk_cyclic_mat(sz) : eval(C) ;
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Note that $C^{-1} = C^T$ and so multiplying $C$ by its transpose yields the identity matrix:

```
> multiply(C, transpose(C));
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The first argument, 1, in $S(1, R)$ indicates that the rotor is being rotated through one position. More generally, the number of positions through which the rotor is rotated does not have to be one. Indeed, it might even vary with time. Let $P^{(j)}$ denote the enciphering permutation matrix after $j$ plaintext characters have been enciphered. If $\mu^{(j)}$ denotes the number of positions through which the rotor is rotated after enciphering the $j$th character, then the next enciphering permutation matrix is given by $P^{(j+1)} = S(\mu^{(j)}, P^{(j)})$, where

$$S(\mu^{(j)}, P^{(j)}) = C^{\mu^{(j)}} P^{(j)} C^{-\mu^{(j)}}$$

A one-rotor machine is completely specified by the rotor permutation matrix, $R$, together with a *rotation function*, $rf$, defining how the rotor is to be rotated. Two examples of simple rotation functions are the following:

```
> # c_e denotes the number of characters enciphered so far
> rf_1 := proc(c_e) 1 end:
> rf_2 := proc(c_e) if irem(c_e, 10) = 0 then
          (5*iquo(c_e, 10)+3 mod 26) else 1 fi end:
> rf_1(3); rf_2(3); rf_1(70); rf_2(70);
                          1
                          1
                          1
                          12
```

Here, $rf_1$ corresponds to rotating the rotor through one position after each character encryption. A one-rotor machine of the form $(R, rf_2)$ also rotates through one position except after every 10 character encryptions, when the rotor is reset in the specified fashion. *irem(p,q)* gives the remainder of the integer division of $p$ by $q$, while *iquo(p,q)* gives the quotient. A Maple procedure for computing $S(\mu, P)$ in a straightforward although not very efficient manner is given in figure 4. This implementation should be regarded simply as a rapid prototype for $S$. Later in the paper, we transform our prototype to obtain a more efficient implementation. In the prototype, the cyclic permutation matrix, $C$, is treated as a global variable.

```
# Given mu and P, S computes S(mu,P) using the
# matrix multiply and matrix transpose commands
# of the linalg package.
S := proc(mu::integer, P::matrix)
  local i, M;
  global C;
  options remember;
  M := P;
  for i from 1 to mu do
    M := multiply(C, multiply(M,transpose(C)))
  od;
  RETURN(M)
end;
```

Figure 4. The *rotor_pkg* function *S*.



Using this command, a rotation through one position of our example rotor from its base position is simulated by:

```
>S(1, R): eval(");
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Here, the double quotes (") refer to the most recently computed variable, which in this case is *S(1, R)*. Hence, *eval(")* fully evaluates *S(1, R)*. From the columns of this matrix, we see that character 1 (*a*) is substituted by character 3 (*c*), character 2 (*b*) by character 5 (*e*), and so on. This is in agreement with figure 1(b). (The reader should note that for Release 5 of Maple V, the percent sign (%) is used instead of double quotes to designate the output from the last executed command.)

In general, the initial setting of the rotor prior to encrypting a message is defined by the number of positions, $\sigma$ say, of the rotor from its base position. The initial enciphering permutation matrix, $P^{(0)}$, is then given by $P^{(0)} = S(\sigma, R)$. Note that since $P^{(j+1)} = S(\mu^{(j)}, P^{(j)})$, it follows that any enciphering permutation matrix has the form $S(\mu, R)$ for some positive integer, $\mu$. Also note that $S(0, R) = S(sz, R) = R$, as verified by the following Maple Boolean expression:

```
> equal(S(sz,R),R) and equal(S(0, R),R);
```

*true*

To simulate the encipherment of a given plaintext message by a rotor machine, we use a *rotor_pkg* function called *rotor_transform* whose definition is given in figure 5. Updating the enciphering permutation matrix, and hence the current substitution cipher, is effected in *rotor_transform* by the pair of assignments *mu_j := rf(j); Pj := S(mu_j, Pj)*. As an example, let

```
> m_stm := [c, e, a, a, d, a, d, b, c]:
```

```
# m_stm is a given message stream; sigma is the indicator;
# rf is the the rotation function and R is the rotor
# permutation matrix.
# rotor_transform(m_stm, sigma, rf, R) returns the
# resulting cipher stream.  The rotor is rotated initially
# through sigma positions from its base position and is
# then rotated through rf(j) positions after the
# j-th character is enciphered.
# This function uses a function, perm, defined
# in figure 6, for converting a permutation
# matrix into the corresponding permutation list.
rotor_transform := proc(m_stm::list, sigma::integer,
    rf::procedure, R::matrix)
  local j, cs, m, mu_j, Pj, p;
  global alphabet, encoding;
  options remember;
  Pj := S(sigma,R);
  cs := [];
  for j from 1 to nops(m_stm) do
    p := perm(Pj);
    m := m_stm[j];
    cs := [op(cs), alphabet[p[encoding[m]]]];
    mu_j := rf(j);
    Pj := S(mu_j, Pj);
  od;
  RETURN(cs)
end;
```

Figure 5. The *rotor_pkg* function *rotor_transform*.



be a given message stream. We simulate enciphering this somewhat meaningless plaintext message using our example one-rotor machine, when the rotor is initialized to its base position and is then rotated through one position for each character enciphered, as follows in which *rf_1* is as defined above:

```
> rotor_transform(m_stm, 0, rf_1, R);
```

$$[a, a, b, d, a, d, b, d, e]$$

For a second illustration, we simulate the encryption process when our simple rotor machine is initially rotated through two positions from its base position and is then rotated through three positions after each successive character is enciphered:

```
> rf_3 := proc(c_e) 3 end:
> c_stm := rotor_transform(m_stm,2,rf_3,R);
```

$$stm := [a, b, d, c, a, b, e, c, d]$$

*rotor_transform* makes use of a subsidiary function, *perm*. Given a permutation matrix, *perm* delivers the corresponding permutation list. The definition of *perm* is given in figure 6.

In order to decipher a message enciphered by a rotor machine, a receiver must possess an identical rotor machine and must also know some *indicator* information. For a one-rotor machine, *(R,rf)*, the indicator is the initial number of positions, $\sigma$, of the rotor in the sender's machine from its base position. Rotor machines such as

```
# Given the matrix representation of a permutation, perm
# generates the list representation of that permutation.
# The rowdim command of the linalg package
# returns the number of rows in its matrix argument.
perm := proc(T::matrix)
  local i, j, vi, p, q;
  q := rowdim(T);
  p := [];
  for i from 1 to q do
    vi := col(T,i);
    j := 1;
    while vi[j] <> 1 do j := j + 1 od;
    p := [op(p), j]
  od;
  RETURN(p)
end;
```

Figure 6. The *rotor_pkg* function *perm*.



the ENIGMA had one-way input but included a *reflector*. Inclusion of the latter meant that if a plaintext character, $c$, enciphered to a character, $x$, when the rotor machine was in a particular position, then in that position the machine would encipher $x$ to $c$. Consequently, given the appropriate indicator, all a receiver had to do to recover the plaintext message from a cryptogram was to mimic the sequence of steps used by the sender. To allow for more general rotor machines, for which $c$ mapping to $x$ in a given position does not necessarily imply $x$ mapping to $c$ in that position, we do not assume the use of a reflector. Rather, we assume that input to a rotor machine is possible in both directions (or, equivalently, that the rotors can be replaced in the machine back-to-front and in reverse order). Now, if $S(\mu, R)$ is an enciphering permutation matrix, then the corresponding deciphering permutation matrix is

$$\begin{aligned}\left(S(\mu, R)\right)^{-1} &= \left(C^{\mu} R C^{-\mu}\right)^{-1} \\ &= C^{\mu} R^{-1} C^{-\mu}\end{aligned}$$

Therefore, to simulate deciphering a cryptogram, *c_stm*, we simply re-apply *rotor_transform* but with *m_stm* replaced by *c_stm* and $R$ replaced by $R^{-1}$. For example, to simulate our rotor machine deciphering the cryptogram *c_stm* = [*a, b, d, c, a, b, e, c, d*] obtained above, we do:

```
> m_stm := rotor_transform(c_stm,2,rf_3,inverse(R));
```

$$m\_stm := [c, e, a, a, d, a, d, b, c]$$

As can be seen, the original message stream has been recovered.

## 4. Program Transfomations

The Maple system has allowed us to produce an executable specification of a one-rotor machine that closely follows our underlying mathematical model. We now transform this prototype to get a more efficient Maple implementation. We

begin by exploiting the structure of the $C$ and $P$ matrices in the implementation of $S(\mu, P)$, where $P$ is any enciphering permutation matrix. The resulting transformation does lead to a moderate gain in efficiency. More importantly, however, the description of how to update an enciphering permutation matrix given in the transformed version of $S$ makes it easier to perform our second program transformation, leading to a dramatic increase in computational speed. Both of our transformations update the implementations of $S$ and *rotor_transform*. In the sequel, therefore, we refer to the original versions of these two functions as $S1$ and *rotor_transform1*, respectively.

From above, successive enciphering permutation matrices are related by the recurrence relation:

$$\begin{aligned} P^{(j+1)} &= S\left(\mu^{(j)}, P^{(j)}\right) \\ &= C^{\mu^{(j)}} P^{(j)} C^{-\mu^{(j)}} \end{aligned}$$

with $P^{(0)} = S(\sigma, R)$. The inverse of $C$ is simply its transpose and the function, $S1$, updates the enciphering permutation matrix directly as follows:

```
M := P;
for i from 1 to mu do
    M := multiply(C,multiply(M,transpose(C))) od;
```

However, we can obtain a more efficient version by exploiting the structure of the $C$ and $P$ matrices. Since each $P$ matrix is just a permutation matrix while $C$ is a cyclic matrix of the form

$$C = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

we can compute $CPC^T$ as follows. Let $P' = CP$. Then $P'$ is given by

$$P' = \begin{bmatrix} \text{row}(P, q) \\ \text{row}(P, 1) \\ \text{row}(P, 2) \\ \vdots \\ \text{row}(P, q-1) \end{bmatrix}$$

where $\text{row}(P, i)$ denotes the $i$th row of $P$. Then, $P'' = CPC^T = P'C^T$ is given by

$$P'' = \left[\text{col}(P', q), \text{col}(P', 1), \text{col}(P', 2), \ldots, \text{col}(P', q-1)\right]$$

where $\text{col}(P, j)$ denotes the $j$th column of $P$. More generally, $P' = C^{\mu}P$ may be computed simply by removing the last $\mu$ rows of $P$ and then stacking them on top of the resulting submatrix. Similarly, $P'' = P'\left(C^T\right)^{\mu}$ may be computed by deleting the last $\mu$ columns of $P'$ and then appending them at the front of the resulting submatrix. The Maple *linalg* package contains commands which enable us to

```
# Given μ and P, S computes C^μ P(C^T)^μ,
# exploiting the structure of the C and
# P matrices.
S2 := proc(mu::integer, P::matrix)
  local q, qm1,M, M1;
  options remember;
  q := rowdim(P);
  M := P;
  if mu = 0 then RETURN(M) fi;
  qm1 := q-mu+1;
  M1 := delrows(M, qm1..q);
  M := stack(row(M, qm1..q), M1);
  M1 := delcols(M, qm1..q);
  M := augment(col(M, qm1..q), M1);
  RETURN(M)
end;
```

Figure 7. A new version of the *rotor_pkg* function *S*.



specify these operations on matrices very easily. A new implementation of *S*, which uses these commands, is given in figure 7. To distinguish this new implementation of *S* from our initial implementation, *S*1, we have called it *S*2. The only change that then needs to be made to the initial implementation of *rotor_transform* is to replace all applications of the original version of *S* by applications of *S*2. We call the resulting version *rotor_transform2*. To get an idea of the gain in efficiency achieved through our first transformation, we consider an example in which we take all 26 lower-case letters as our alphabet. The fixed connections of the rotor in an appropriate one-rotor machine are then described by some $26 \times 26$ permutation matrix, *R*. Suppose we then want to simulate using such a machine to encipher the title of this paper, ignoring spaces between words. Assume that the rotor is to start in its base position and that the rotor is to be rotated through three positions after each letter has been enciphered. Then, on the computer being used (a DEC alpha 600 workstation), *rotor_transform* took approximately 129 seconds to encrypt this message using the original version of *S* but only approximately 22 seconds using the transformed version of *S* (see figure 11). Similarly, deciphering the resulting cryptogram takes approximately 141 seconds with the original version of *S* but only 26 seconds using the transformed version. Of course, these times vary slightly from run to run, but on our machine the transformed version of *rotor_transform* typically runs approximately six times faster than does the original version.

We note that since *S*1 is the only *rotor_pkg* command that uses the cyclic matrix, *C*, if we replace *S*1 in *rotor_pkg* by *S*2, then the function *mk_cyclic_mat* is no longer needed.

Our second transformation is based on the fact that we do not need to use permutation matrices explicitly. Using matrix similarity transformations to represent changes in the enciphering permutation when the rotor is rotated is a useful device while developing the underlying mathematical model. However, *rotor_transform*, which is the only *rotor_pkg* function that uses *S(mu, P)*, first uses the current enciphering/deciphering permutation matrix to generate a vector representation. It is this vector representation which is then used to encipher/

```
rotor_transform3 := proc(m_stm::list, sigma::integer,
    rf::procedure, p_R::list)
  local j, cs, m, mu_j, p_j, p;
  global alphabet, encoding;
  options remember;
  p_j := S3(sigma, p_R);
  cs := [];
  for j from 1 to nops(m_stm) do
    m := m_stm[j];
    cs := [op(cs), alphabet[p_j[encoding[m]]]];
    mu_j := rf(j);
    p_j := S3(mu, p_j);
  od;
  RETURN(cs)
end;
```

Figure 8. Version 3 of the *rotor_pkg* function *rotor_transform*.



decipher the current character. The *rotor_pkg* function, *perm*, which generates the required permutation vector from a given permutation matrix, has run-time complexity $O(q^2)$, where $q$ is the size of the alphabet. By representing enciphering permutations directly as vectors, we would remove the need to use *perm*. We represent permutation vectors by Maple lists. Note that we can use square bracket notation to access elements of such lists, e.g. $ls[i]$. This is, in fact, more efficient than using $op(i, ls)$ as the latter results in the evaluation of the entire list.

Assuming therefore that enciphering permutations are to be represented by Maple lists rather than by matrices, our third implementation of *rotor_transform* (referred to here as *rotor_transform3*) is as given in figure 8. *rotor_transform3* uses a function, $S3$, which updates enciphering/deciphering permutation lists. We must now derive the definition of $S3$.

If $p$ is a permutation vector of length $q$ and $P$ is the corresponding $q \times q$ permutation matrix, then $p[j] = i$ *iff* column $j$ of $P$ has its unit element in row $i$. Now suppose that $P$ and $P'$ denote two successive $q \times q$ enciphering permutation matrices, where $P'$ is the result of applying $S2$ to $\mu$ and $P$, and let $p$ and $p'$ be the corresponding permutation vectors. From the definition of $S2$, $P'$ is derived from $P$ by first deleting the last $\mu$ rows of $P$ and placing these rows on top of the remaining submatrix. Let us call the resulting intermediate permutation matrix $\hat{P}$ and its corresponding permutation vector $\hat{p}$. Suppose column $j$ of $P$ has its unit element somewhere in the first $q-\mu$ rows, say in row $i$, $1 \le i \le q-\mu$. Since row $i$ in $P$ is pushed down $\mu$ rows in $\hat{P}$, column $j$ of $\hat{P}$ has its unit element in row $i+\mu$. Now suppose that column $j$ of $P$ has its unit element somewhere in the final $\mu$ rows of $P$, say in row $i$, $q-\mu+1 \le i \le q$. In this case, row $i$ in $P$ is shifted up $q-\mu$ rows in $\hat{P}$. Hence, column $j$ of $\hat{P}$ has its unit element in row $i-q+\mu$. In terms of the permutation vectors $p$ and $\hat{p}$, we have therefore

$$\hat{p}[j] = \begin{cases} p[j] + \mu & 1 \le p[j] \le q-\mu \\ p[j] - q + \mu & \text{otherwise} \end{cases}$$

Returning now to $S2$, we see that the updated enciphering permutation matrix, $P'$, is obtained from the intermediate permutation matrix, $\hat{P}$, by deleting the last $\mu$

```
# To determine the new enciphering permutation when the rotor
# is rotated through mu positions, given the current
# enciphering permutation, p.
S3 := proc(mu::integer, p::list)
local q, j, new_p, qm;
options remember;
if mu = 0 then RETURN(p) fi;
q := nops(p);
qm := q - mu;
new_p := p; # To ensure new_p is initialized to a list of length q.
for j from 1 to q do
  if p[j] <= qm then
    new_p[j] := p[j] + mu
  else
    new_p[j] := p[j] - qm
  fi
od;
new_p := [op(new_p[-mu..-1]), op(new_p[1..qm])];
                  # concatenate sublist consisting
                  # of last mu elements with that of
                  # the first q-mu elements.
RETURN(new_p)
end;
```

Figure 9. Definition of the *rotor_pkg* function *S3*.



columns of $\hat{P}$ and placing these columns at the front of the resulting submatrix. In this case, the position of each column is changed but the position of the unit element within that column is unaltered. To obtain the enciphering permutation vector, $p'$, from the intermediate permutation vector, $\hat{p}$, therefore, we simply transfer the final $\mu$ elements of $\hat{p}$ to the front of the vector.

The definition of $S3$ based on the above discussion is given in figure 9. Note that if *ls* is a Maple list then $ls[1 \ldots k]$ is the sublist consisting of the first $k$ elements of *ls*, while $ls[-k \ldots -1]$ is the sublist consisting of the last $k$ elements. Also note that in $S3$ it is more efficient first to create the sublists $new\_p[-mu \ldots -1]$ and $new\_p[1 \ldots qm]$ and then to convert these sublists into sequences using the *op* command than to use *op* directly with the appropriate ranges (e.g. $\mathrm{op}(1 \ldots qm, new\_p)$).

From above, if $S(\mu, R)$ is an enciphering permutation matrix, then $S(\mu, R^{-1})$ is the corresponding deciphering permutation matrix. Now, if $p_R$ is the permutation vector corresponding to the rotor permutation matrix, $R$, then $p_R^{-1}$ is the permutation vector corresponding to $R^{-1}$. A cryptogram, *c_stm*, may thus be deciphered by evaluating *rotor_transform3(c_stm,* $\sigma$*, rf,* $p_R^{-1}$*)*, where $\sigma$ is the given indicator and *rf* is the rotation function of the given machine. The definition of a function from *rotor_pkg* for inverting a permutation vector is given in figure 10.

The effect of our transformations on the computational speed of *rotor_transform* is illustrated in figure 11. This figure records a Maple session in which we timed how long each of our versions of *rotor_transform* took to encipher the title of this paper. As can be seen, the speed-up for version 3 is considerable. All of the

```
# To determine the inverse of a given permutation vector
inv_perm := proc(p::list)
local i, q, inv_p;
options remember;
q := nops(p);
inv_p := p;
for i from 1 to q do
  inv_p[p[i]] := i
od;
RETURN(inv_p)
end;
```

Figure 10. Definition of the *rotor_pkg* function *inv_perm*.



timing results given in this paper were obtained using a DEC alpha 600 workstation.

## 5. *N*-rotor machines

We now generalize our treatment of rotor machines to those posssessing $N \geq 1$ rotors. Each rotor, $i$, in an $N$-rotor machine is specified by a pair, $(R_i, rf_i)$, where $R_i$ is a rotor permutation matrix defining the fixed internal connections of that rotor and $rf_i$ is a rotation function defining how the rotor is rotated after each character encryption/decryption. Thus, $rf_i(j)$ gives the number of positions through which the $i$th rotor is rotated after enciphering its $j$th input. Let $R$ now denote the list of rotor permutation matrices. The initial setting of the machine prior to encrypting a message, i.e. the indicator, is defined by the number of positions, $\sigma_i$, of each rotor, $i$, from its base position. Let $\sigma$ now denote the list of $\sigma_i$s. The corresponding initial enciphering permutation matrix is then given by $S(\sigma, R)$, where

$$S(\sigma, R) = C^{\sigma_N} R_N C^{-\sigma_N} C^{\sigma_{N-1}} R_{N-1} \cdots R_1 C^{-\sigma_1}$$

$$= S(\sigma_N, R_N) S(\sigma_{N-1}, R_{N-1}) \cdots S(\sigma_1, R_1).$$

Thus, the initial enciphering permutation matrix is the product of the initial enciphering permutation matrices for the individual rotors. Here we are assuming that the plaintext message is entered at rotor 1, the output from which passes to rotor 2, etc. Now let

$$P^{(j)} = [P_1^{(j)}, P_2^{(j)}, \ldots, P_N^{(j)}]$$

denote the list of enciphering permutation matrices for the individual rotors after $j$ plaintext characters have been enciphered, and let

$$\mu^{(j)} = [rf_1(j), rf_2(j), \ldots, rf_N(j)].$$

Then

```
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace
> read(rotor_pkg):
> sz:=26:
> alphabet:=[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]:
> for cd from 1 to sz do encoding[alphabet[cd]]:=cd od:
> C:=mk_cyclic_mat(sz):
> pR:=[3,7,22,15,1,9,4,24,19,17,5,11,25,2,16,12,23,14,26,8,13,21
  ,6,18,10,20]:
> R:=mk_rotor(sz, pR):
> rf:=proc(x) 3 end:
> sigma:=0:
> m_stm:=[s,p,e,c,i,f,y,i,n,g,a,n,d,s,i,m,u,l,a,t,i,n,g,r,o,t,o,r,
  m,a,c,h,i,n,e,s,u,s,i,n,g,m,a,p,l,e]:
> st:=time():c1_stm:=rotor_transform1(m_stm,sigma,rf,R):time()-st;
                                  129.482
> c1_stm;
[z,b,p,q,r,l,v,w,j,j,j,k,r,v,p,a,f,x,l,u,k,g,n,t,g,c,p,s,j,w,i,o,o,u,b,o,a,r,t,p,m,
  a,w,v,s,y]
> st:=time():m1_stm:=rotor_transform1(c1_stm,sigma,rf,inverse(R)):
  time()-st;
                                  141.014
> m1_stm;
[s,p,e,c,i,f,y,i,n,g,a,n,d,s,i,m,u,l,a,t,i,n,g,r,o,t,o,r,m,a,c,h,i,n,e,s,u,s,i,n,g,
  m,a,p,l,e]
> st:=time():c2_stm:=rotor_transform2(m_stm,sigma,rf,R):time()-st;
                                  22.029
> c2_stm;
[z,b,p,q,r,l,v,w,j,j,j,k,r,v,p,a,f,x,l,u,k,g,n,t,g,c,p,s,j,w,i,o,o,u,b,o,a,r,t,p,m,
  a,w,v,s,y]
> st:=time():m2_stm:=rotor_transform2(c2_stm,sigma,rf,inverse(R)):
  time()-st;
                                  26.536
> st:=time():c3_stm:=rotor_transform3(m_stm,sigma,rf,pR):time()-st;
                                  .040
> st:=time():m3_stm:=rotor_transform3(c3_stm,sigma,rf,inv_perm(pR)):
  time()-st;
                                  .043
> m3_stm;
[s,p,e,c,i,f,y,i,n,g,a,n,d,s,i,m,u,l,a,t,i,n,g,r,o,t,o,r,m,a,c,h,i,n,e,s,u,s,i,n,g,
  m,a,p,l,e]
```

Figure 11. Timings for the three versions of *rotor_transform*.

$$P^{(j+1)} = S_N(\mu^{(j)}, P^{(j)})$$
$$= [S^{(N)}(\mu_1^{(j)}, P_1^{(j)}), S^{(N)}(\mu_2^{(j)}, P_2^{(j)}), \ldots, S^{(N)}(\mu_N^{(j)}, P_N^{(j)})].$$

It follows that we can simulate an $N$-rotor machine by using in an iterative fashion any of the versions of $S$ described above. We will use $S3$, which is our most efficient version. The resulting *rotor_pkg* function is called *S_N*. Since $S3$ represents the current enciphering permutation of a rotor by a Maple list rather than by a matrix, the second argument of *S_N* is a list of lists, *ps*, each element of

```
# To determine the new list of enciphering permutations
# when each rotor, i, is rotated through mu[i] positions.
# ps is the current list of enciphering permutations.
S_N := proc(mu::list, ps::list)
local i, new_ps, N;
options remember;
N := nops(mu);
new_ps := [seq(S3(mu[i], ps[i]), i = 1..N)];
RETURN(new_ps)
end;
```

Figure 12. Definition of the *rotor_pkg* function *S_N*.



```
# m_stm is the message stream;
# sigma is the indicator (a list of non-negative integers);
# rf is a list of rotation functions and rp is a list of
# rotor permutation vectors such that (rp[i], rf[i])
# specifies the i-th rotor in an N-rotor machine.
# rotor_transform_N(m_stm, sigma, rf, rp) gives the
# encrypted stream corresponding to m_stm.
# For each i, rotor i is rotated initially through sigma[i]
# positions from its base position and is then rotated
# through rf[i](j) positions after each successive
# character is enciphered or deciphered.
rotor_transform_N := proc(m_stm::list, sigma::list,
   rf::list, rp::list)
  local i, j, cs, m, p_j, ps_j, N, mu;
  global alphabet, encoding;
  options remember;
  N := nops(rf);
  ps_j := S_N(sigma, rp);
  p_j := compose_perms(ps_j);
  cs := [];
  for j from 1 to nops(m_stm) do
    m := m_stm[j];
    cs := [op(cs), alphabet[p_j[encoding[m]]]];
    mu := [seq(rf[i](j), i = 1..N)];
    ps_j := S_N(mu, ps_j);
    p_j := compose_perms(ps_j);
  od;
  RETURN(cs)
end;
```

Figure 13. Definition of *rotor_transform_N*.

which is the list representation of a permutation vector. The definition of *S_N* is given in figure 12 while that of the function, *rotor_transform_N*, which uses *S_N* to encipher a message stream, is given in figure 13. The latter makes use of a function called *compose_perms*. Given a list of permutation vectors, $p_1, p_2, \cdots, p_N$, say, *compose_perms* computes the composition

$$ps = p_N \circ p_{N-1} \circ \cdots \circ p_1$$

```
# To compose a sequence of permutations
compose_perms := proc(ps::list)
local i, j, p, q, N;
options remember;
N := nops(ps);
p := ps[1];
q := nops(p);
for i from 2 to N do
  p := [seq(ps[i, p[j]], j = 1..q)]
od;
RETURN(p)
end;
```

Figure 14. Definition of *compose_perms*.

```
# To determine the inverse of each permutation
# vector in a list, ps, of such vectors, and return
# the list of inverses in reverse order.
rev_inv_perms := proc(ps::list)
local i, N, r_invps;
options remember;
N := nops(ps);
r_invps := [seq(inv_perm(ps[-i]),i = 1..N)];
RETURN(r_invps)
end;
```

Figure 15. Definition of *rev_inv_perms*.



If $p_i$ represents the current enciphering permutation of rotor $i$ in an $N$-rotor machine, then *ps* represents the current enciphering permutation of the entire machine. The definition of *compose_perms* is given in figure 14.

A cryptogram, *c_stm*, enciphered using an $N$-rotor machine may be deciphered using an identical machine whose rotors are initialized in precisely the same way as were the rotors in the enciphering machine. Recall that we are assuming that input to a rotor machine is possible from both ends and that an encrypted message is input from the opposite end to that used to input the original plaintext message. To simulate deciphering *c_stm*, therefore, we call *rotor_transform_N* with *c_stm* in place of *m_stm*, with the list of $\sigma_i$s and the list of rotation functions both in reverse order, and with the list of rotor permutation vectors, *rp*, replaced by the corresponding list of inverse permutation vectors, again in reverse order. For convenience, *rotor_pkg* includes a function, *rev_ls*, for reversing a list in the manner indicated earlier in the paper. A separate function, *rev_inv_perms*, is used to invert each of the permutations in a list and reverse their order. The definition of this function is given in figure 15.

The use of *rotor_transform_N* in a Maple session is illustrated in figure 16. A 3-rotor machine is specified and its use to encipher the title of this paper, and then to decipher the resulting cryptogram, is simulated.

We end our discussion of rotor machines by briefly considering the most famous of them—the ENIGMA machine. We have seen that a rotor machine may

```
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace
> read(rotor_pkg):
> sz:=26:
> alphabet:=[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z]:
> for cd from 1 to sz do encoding[op(cd, alphabet)]:= cd od:
> pR1 := [3,7,22,15,1,9,4,24,19,17,5,11,25,2,16,12,23,14,26,8,13,21,
  6,18,10,20]:
> pR2 := [23,18,2,12,11,9,3,25,19,4,5,24,20,22,6,15,17,14,26,8,13,
  21,16,7,1,10]:
> pR3 := [20,12,7,19,11,23,5,21,6,15,14,16,2,18,3,26,10,24,1,25,9,
  4,22,8,13,17]:
> p_Rs := [pR1,pR2,pR3]:
> rf_1 := proc(x) 1 end:
> rf_2 := proc(x) if irem(x, 10) = 0 then (5*iquo(x, 10) mod 26)
  else 1 fi end:
> rf_3 := proc(x) 3 end:
> rf_s := [rf_1,rf_2,rf_3]:
> sigmas := [0,1,2]:
> m_stm:=[s,p,e,c,i,f,y,i,n,g,a,n,d,s,i,m,u,l,a,t,i,n,g,r,o,t,o,r,
  m,a,c,h,i,n,e,s,u,s,i,n,g,m,a,p,l,e]:
> st:=time():c_stm:=rotor_transform_N(m_stm,sigmas,rf_s,p_Rs):
  time()-st;
                                .162
> c_stm;
[s,l,c,l,h,v,l,v,u,k,e,e,p,y,r,p,a,p,v,y,a,f,h,f,q,h,t,q,t,u,d,r,d,g,k,v,q,x,h,t,a,
  n,p,f,d,r]
> st:=time():m1_stm:=rotor_transform_N(c_stm,rev_ls(sigmas),
  rev_ls(rf_s),rev_inv_perms(p_Rs)):time()-st;
                                .184
> m1_stm;
[s,p,e,c,i,f,y,i,n,g,a,n,d,s,i,m,u,l,a,t,i,n,g,r,o,t,o,r,m,a,c,h,i,n,e,s,u,s,i,n,g,
  m,a,p,l,e]
```

Figure 16. An example of the use of *rotor_transform_N*.

be completely specified by giving for each rotor, $i$, the rotor permutation (either as a matrix, $R_i$, or as a vector, $rp_i$), and the rotation function, $rf_i$. Since $rf_i$ may be defined in any way we like, the model presented in this paper is more general than actual rotor machines like the ENIGMA. Typical versions of the ENIGMA possessed three rotors whose motion was odometer-like. The first rotor in such a machine moved through one position after enciphering/deciphering each character. The second rotor moved through one position only when the first rotor passed a certain point. We may take this point to be the first rotor's base point. Similarly, the third rotor moved through one position only when the second rotor passed through its base position. Maple definitions of the rotation functions for an ENIGMA style rotor machine are as follows, in which `sigma` denotes the indicator:

```
rf_1 := proc(x) 1 end:
rf_2 := proc(x) if irem(x + sigma[1], 26) = 0 then 1
                  else 0 fi end:
```

```
rf_3 := proc(x) if irem(iquo(x + sigma[1], 26)
                    + sigma[2], 26) = 0 then 1 else 0 fi end:
rf:=[rf+1,rf_2,rf_3]:
```

(For mechanical reasons, the rotational movements of the rotors in the ENIGMA itself were slightly different from those specified here.) In principle, a rotor machine with sufficiently many rotors should be difficult to attack, even if a cryptanalyst knows the rotor permutations and the rotation functions. The ENIGMA was cracked due to the use of a poor method for communicating the indicator. Such key distribution problems are the drawback of most classical cryptosystems.

## 6. Conclusion

As shown in this paper, symbolic computation languages such as Maple can be used to produce rapid prototypes of software systems and executable specifications of hardware systems. By then performing program transformations, reasonably efficient implementations can be obtained. We have used this approach to develop a very general executable model of a rotor machine. A similar approach could be used to produce executable specifications of hardware devices such as systolic arrays.



## References

[1] Hearn, A. C., 1968, in M. Klerer and J. Reinfelds, editors, *Interactive Systems for Experimental Applied Mathematics*, pp. 79–90 (Academic Press) New York.
[2] Martin, W. A., and Moses, J., 1970, Mathlab (MACSYMA), MAC Programmning report V, Massachusetts Institute of Technology, A. I. Lab., Cambridge, MA, USA, July 1969 to December 1970.
[3] Waterloo Maple Inc., 1995, *The Maple Learning Guide.*
[4] Waterloo Maple Inc., 1995, *The Maple Programming Guide.*
[5] Wolfram, S., 1991, *Mathematica: a System for Doing Mathematics by Computer*, second edition (Allison-Wesley).
[6] Christensen, S. M., 1994, *Comput. Phys.*, **8**, 308–315.
[7] Churchhouse, R. F., 1991, *IMA Bull.*, **27**(7), 129–137.
[8] Churchhouse, R. F., 1993, *IMA Bull.*, **29**(9), 129–135.
[9] Bagnall, A. J., McKeown, G. P., and Rayward-Smith, V. J., 1997, The cryptanalysis of a three rotor machine using a genetic algorithm, in *Proceedings of ICGA 97.*
[10] Andelman, D., and Reeds, J., 1982, *IEEE Trans. Inf. Theory*, **4**, 578–584.