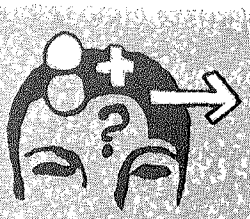

Techniques

R. M. GRAHAM, Editor

The Performance of a System for Automatic Segmentation of Programs Within an ALGOL Compiler (GIER ALGOL)

PETER NAUR
Regnecentralen, Copenhagen, Denmark

The GIER ALGOL compiler makes use of an automatic system for handling the transfers of program segments from the drum store to the core store at program execution time. The logic of this system is described. The performance of the system is discussed, primarily on the basis of execution times related to two specific programs. The discussion concludes with an assessment of the potential gains of various ways of improving the system.

Introduction

In programming for a machine with a nonhomogeneous store, such as a core store and a backing drum, the handling of the transfers of program and data between the two media is usually a major problem. In designing an ALGOL compiler for such a machine this problem is added to all the other problems of the compiler. It soon becomes clear that the problems of program and data are essentially different, for two reasons. On the one hand, the programmer knows far more about the most suitable allocation of data than of program, because he knows the volume of the data, while his knowledge of the size of the translated program is usually very crude. On the other hand, an automatic handling of program transfers is much simpler than the corresponding handling of data because the program is unaltered during execution, at least in machines having reasonable addressing facilities.

In planning the ALGOL compiler for the GIER, a machine having 1024 words of core store and 12,800 words on drum, it was therefore decided to include an automatic segmentation of the program, while the handling of data on the drum is left to the programmer. This decision was based on rather crude estimates as to the structure of actual programs, but has proved very successful in practice. In fact, the programming for the GIER machines now running is done almost exclusively in ALGOL. The time is therefore ripe for another discussion of the problem, using the experience of the GIER ALGOL system as the starting point.

The present paper contains first a brief description of the logic of the segmentation scheme. This is followed by a report and a discussion of some experiments on the performance of the scheme in a few representative programs.

The Segmentation Scheme of GIER ALGOL

The principal design of the GIER ALGOL compiler has already been described [1]. In particular, the program storage allocation scheme is described in Sections 3.9 to 3.12 of [1]. The basic features of the scheme are the following.

1. The compiler has divided the translated program into a sequence of segments each held on one track of the drum. A program track of 40 machine words will on the average hold about 60 machine instructions. The division into segments is done by filling the translated program into tracks without regard to the block, or other, structure of the source program.

2. The compiler has prepared each segment to be executed from any location of the core store and during execution the segment will make no assumptions with regard to the presence of other segments in the core store.

3. The transfer of segments from the drum to the core store is done by a fixed administration which is kept permanently in the core store at run time. This administration will keep a list of the track numbers of the segments currently present in the core store. Whenever the control is transferred from one segment to another the administrative routine will examine this list, thus avoiding unnecessary transfers of program segments from the drum.

4. The amount of core storage available for program segments is that which is left over when the locations necessary for the currently declared variables have been reserved. Consequently, the number of segments held in the core store will generally change during the execution of the program.

5. When the transfer of control to a new segment forces a segment in the core store to be cancelled, the segment is chosen for cancellation which has been left unused for the longest time.

The administration of this logic is shown in the ALGOL procedure given in Appendix 1.

Presented at the International Symposium on Data Processing Machines in Prague, September 1964.

To illustrate the logic Figure 1 shows the use of the core store at two stages during the execution of a representative program. The words of the store are identified by their absolute addresses, running from 0 to 1023. A program segment place needs 41 words, one of which is used to hold the *PRIORITY* of the track while the rest hold the instructions. The first few locations of the store are reserved

Locations	*Situation 1*		*Situation 2*	
	Track		*Track*	
6–46	288		288	
47–87	289		304	5 Program Segments
88–128	308		258	
129–169	305	8 Program Segments	305	
170–210	290		257	
211–251	258			
252–292	304			Variables of the Program
293–333	257			
334–835	Variables of the Program			
836–1023	Fixed Administration		Fixed Administration	

FIG. 1. Allocation of core storage

for internal purposes, so the first segment place uses the locations 6 to 46, the second the locations from 47 to 87, etc. The system will never work with less than two segment places, and the maximum capacity is 20 places. The fixed administration, including the table of the track numbers of the available program segments (SEGMENT TRACK) and several other routines which will not be described here, uses the locations from 836 to 1023. The stack of variables of the program starts at location 835 and will at any time reserve as many locations as are needed according to the declarations of the program. The largest number of variables which can be accomodated is therefore 746. As shown in detail below it is, however, unwise to go that far, for reasons of the execution time. The two illustrations of the use of the storage should serve to impress on the reader the fact that the system at all times tries to make the best possible use of the available core store.

As a basis for the further discussion of the scheme some figures on the execution times of some of the basic tasks are necessary. These are given in Table 1, extracted from [2, App. 3].

The range of times given for the transfer of control to a track already in the core store reflects the time of search through the *SEGMENT TRACK* table. Clearly the time depends on how soon the track is found. Details of this in specific cases are given below.

The Performance of the System

The present study of the performance is concerned with the time spent on performing the various administrative tasks involved. The most important of these are the time used for transferring drum tracks and the time used for handling the transfer of control among such tracks which are already present in the core store. The purpose of the study is to determine the relative importance of the various tasks in realistic cases, with a view to possible improvements of the system. The improvements considered are those which would entail only relatively minor changes of the algorithms or the machine. Changes which involve the adoption of a basically different approach, such as a reliance of information supplied by the programmer, are not considered here.

As the first problem, let us try to get an idea of the significance of the drum transfers for the overall execution time. This will be important in defining the circumstances under which other factors will have to be studied. Clearly the importance of drum transfers will depend on the number of segment places available in the core store. It is therefore natural to study the execution time of a given program as a function of the available number of drum tracks. A realistic way of determining this function is to perform timing experiments on the machine. A fairly simple way of doing this is the following. We embed the given program in two blocks of the form:

```
begin integer p;
input (p);
  begin array A[1: p];
  . . . Here we write the given program
  end;
end;
```

TABLE 1. EXECUTION TIMES

	Milliseconds
Transfer of track from drum	21
Transfer of control to track already in core store 0.7	1.6
Reference to subscripted variable, 1 subscript	0.9
Reference to subscripted variable, 2 subscript	1.2
Floating point addition	0.12
Floating point multiplication	0.18
Address coincidence test (search comparison)	0.044

TABLE 2

Segment places (1)	*Inversion*				*INTEGRATION*			
	Drum Transfers (2)	*Segment Transitions* (3)	*Search Comparisons* (4)	*Run time (sec)* (5)	*Drum Transfers* (2)	*Segment Transitions* (3)	*Search Comparisons* (4)	*Run Time (sec)* (5)
2	96	645	971	9.2	822	477	802	21.1
3	66	675	1395	8.4	700	588	1256	18.6
4	67	675	1542	8.4	534	755	1979	15.2
5	65	676	2230	8.4	249	1041	3335	9.1
6	11	731	2549	7.3	139	1148	4607	6.9
7	10	731	2755	7.5	129	1159	5060	6.5
8	12	731	3056	7.4	99	1189	7010	6.0
9	10	731	3484	7.5	64	1223	5951	5.3
10	9	732	4176	7.4	10	1277	5117	4.0
11	8	733	2519	7.5	9	1278	5794	4.1
12	7	734	4341	7.5	7	1280	7310	4.0
13	8	733	3237	7.3	4	1284	6071	4.1
14	9	732	5882	7.3	1	1286	6671	4.0
15					6	1281	8061	4.1
16					9	1278	8400	4.2

By varying p we can at will reserve any desired fraction of the core store.

Test Programs and Results

Experiments along these lines have been made for two programs, chosen to represent two opposite extremes with respect to loop structure, as follows:

Inversion Program—the inversion of a matrix of order 10, using a slightly modified version of the *gjr* procedure of Rutishauser [3].

Integration Program—the evaluation of a double integral by means of a Romberg procedure used recursively [4].

More details of the programs are given in Appendix 2.

In each case the value of p was varied in steps of 41, corresponding to the amount of store required by one segment, until the capacity of the core store was exceeded. The last successful run then corresponds to 2 segment places. The run times were obtained by the use of a stopwatch and so are only accurate within a few tenths of a second.

In order to facilitate the interpretation of the run-times obtained in this way, runs of the same programs were performed in which the fixed administration was modified slightly in such a manner that counts of the following three actions were obtained: (1) drum transfers, (2) transfers of control to segment already in core, and (3) tests for coincidence of a searched track number and items in the *SEGMENT TRACK* table. Counts (2) and (3) together allow a calculation of the exact time used for transfers of control to segments in the core store.

The results of the experiments are given in Table 2 and are represented in Figures 2 and 3. They are discussed in the following sections.

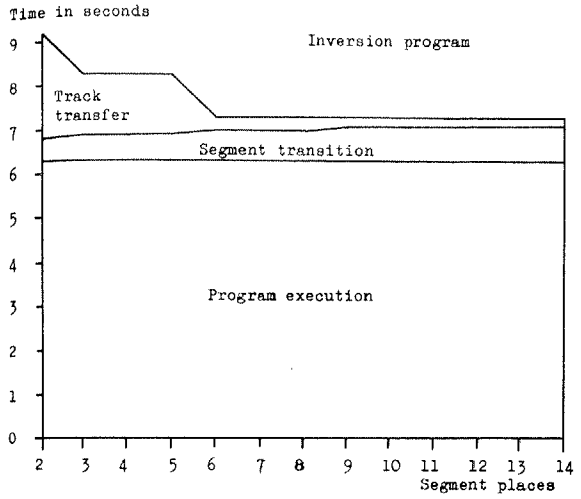

FIG. 2

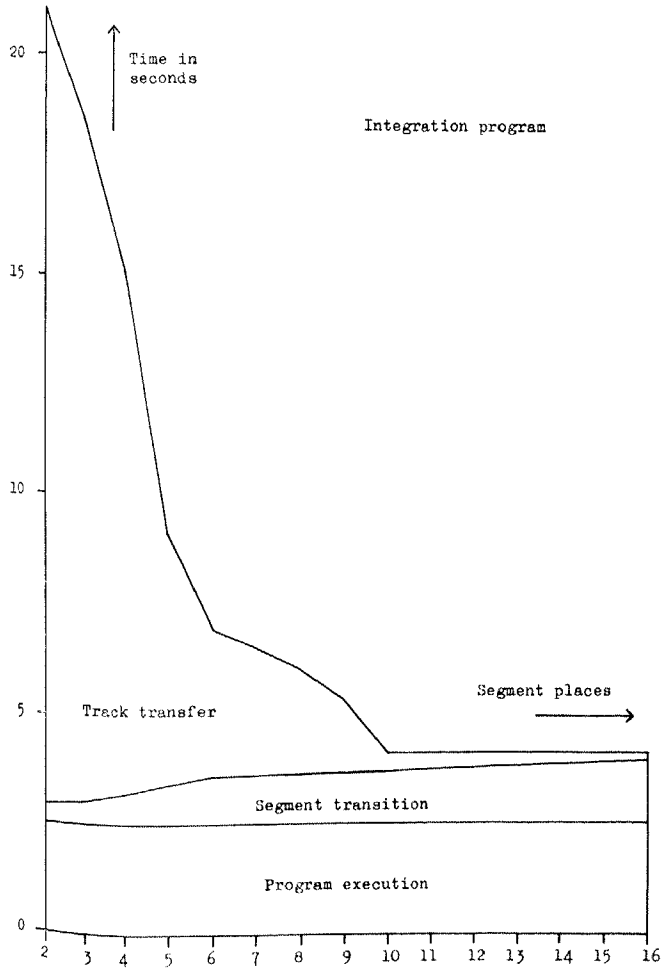

FIG. 3

Drum Transfer Times

The differences in the essential features of Figures 2 and 3 can be readily explained in terms of the structure of the two programs as follows. The innermost loop of the inversion program is the statement:

for j := 1 **step** 1 **until** n **do** $a[i, j] := a[i, j] + z \times b[j]$

With $n = 10$ this loop is executed 1000 times. This loop is so short that it will be held in one or at most two segments. Consequently, even when there are only two segment places available in the core store this loop can be accomodated. In the particular experiment it is apparent that the loop has been placed on only one track by the translator, since otherwise the number of transfers of control across segments would be greater than 2000. This explains that the program will run with no substantial increase of run time due to drum transfers, even when there are only two segment places available. By contrast, the innermost loop of the integration program is composed of the following program pieces: from procedure Romberg:

```
for k := 1 step 1 until max ord do
  begin
  x := j/p;  x := x × a + (1−x) × b;  f0 := f;
  I2 := sum + f0;
  error := error + (if abs (f0) > abs(sum)
    then sum − (I2−f0) else f0 − (I2−sum));
  sum := I2
  end summation of function values and errors;
```

and actual parameter corresponding to f:

if $w = 0$ **then** 0 **else** $(-ln(w))\uparrow(-n)$

This loop comprises four different parts: the Romberg procedure, the actual parameter, the *ln* code and the power code. These are all stored in different places on the drum and depending on the segmentation done by the translator will use from four to six different tracks. This explains the drastic rise of drum transfer time for less than six available segment places.

It should be realized that the drum transfer times just discussed in no way can be used as an argument against the automatic segment administration scheme. They are entirely a function of the limited size of the machine and would arise when the core store is filled almost to the brim with variables no matter which administration scheme were used. The way to avoid them is to organize the program so as to keep the number of variables in the core store below a certain limit.

The important result of the present section is that the drum transfer times are important only when the program is squeezed rather tightly by variables, while there is a wide range of normal situations in which these times are insignificant. It therefore makes sense to continue the discussion of these normal situations.

Segment Transition Times

We now discuss the normal situations, as represented by the inversion program running with six segment places or more, and by the integration program running with 10 segment places or more. In these situations we find that the segment transition times amount to 11 percent and 32 percent of the total execution time in the two programs. However, these times are quite sensitive to otherwise insignificant changes of the programs, which may move a segment transition in or out of the innermost loop of the program. As an example, suppose that a segment transition had been placed right in the innermost loop of the inversion program. This would cause an extra 2000 segment transitions, using about 2.2 seconds. The execution time would be 9.6 seconds and the segment transition time would be 31 percent of this.

More generally, the shorter the innermost loop contributing to the bulk of the execution time, the more sensitive will the program be to differences of segmentation and the more significant may the transition time become. The extreme case which is still of practical interest is probably something like the following:

$$\textbf{for } i := j \textbf{ while } A[i] \neq y \textbf{ do } j := j+1;$$

The loop time of this is about 1.6ms. A segment transition placed in this might in the extreme case contribute an extra 3.2ms, thus slowing the execution down by a factor of 3. The other extreme is the case of a long loop including extensive calculations, without calls of procedures or formal names. In this case there will be up to 80 floating operations between each segment transition. This will give a constant contribution of transition times of about 10 percent, independent of the segmentation.

Although not alarming, these figures invite a consideration of possible ways to reduce them. Three such possibilities are described here. The first possibility is to add suitable logic to the translator for detecting a suitably chosen class of short innermost loops and making sure that no such loop is placed across a segment transition, if necessary by wasting space on a drum track. This approach would be fairly simple to implement, in particular in a multipass translator like GIER ALGOL. However, it is of rather limited value, since it would only help to avoid the outstandingly bad cases while wasting space on drum tracks in many unimportant cases; not every short innermost loop is a significant contributor to the execution time.

A second possibility would be to include in the run-time administration a periodic analysis of the program segment configuration with a possibility of replacing segment transitions within available segments by direct jumps. This might turn out to be a simpler matter than might perhaps appear at first sight. However, again the advantages are doubtful, since the detection of a candidate configuration for jump replacement will depend on this configuration having already run for many cycles, which may well be too late.

A third possibility would be to add a new instruction to the hardware logic of the machine to take care of the segment transition. If this were done the transition might be speeded up by a factor of about 6, which means that the problem would be solved. Within GIER ALGOL the cost of this solution would be high, since both hardware and software would have to be changed. However, in a new machine this would be the obvious solution.

Effect of the Program Storage Cleanup

The figures of Table 2 allow an estimate of the effect of the program storage cleanup performed at every 510 segment transitions. The cleanup cancels the segments stored closest to the stack as long as they have not been used during the last 256 segment transitions. This shortening of the segment table will make itself felt during the execution of loops which can easily be held in the available segment places, by reducing the size of the *SEGMENT TRACK* table to be searched at every segment transition. To obtain a direct expression of this effect we may compare the average number of the actually performed search comparisons per segment transition with the number to be expected if all free segment places were used, $(p+1)/2$, where p is the number of places. The relation between these two numbers is shown in Figure 4. This figure shows clearly that when there are more than a certain number of available segment places the number of search comparisons is usually well below the expected number.

To obtain the effect on the total execution time we may note that the saving realized by the cleanup technique amounts to roughly a third to a half of the total time used for the search comparisons. On the basis of Tables 1 and 2 this is found to be a few percent of the

total execution time. The saving must be weighed against the costs of the feature. These arise partly from the periodic cleanup process itself, partly from the drum transfers which would have been avoided if the segments had not been cancelled. The first of these amounts to less than 2 milliseconds every 510 segment transitions and so is negligible. The second is highly dependent on the program and may range from nothing to an amount which will swamp the complete saving.

It must therefore be concluded that the program storage cleanup feature is a doubtful advantage which probably might as well be omitted.

Varying the Segment Size

All the experiments reported above refer to the same segment size, 40 words. In general the segment size should be considered to be a parameter chosen in some optimum manner. We have no experimental results bearing directly on this matter, but must confine ourselves to the following discussion.

An upper limit to the segment size is imposed by the necessity of being able to accommodate innermost loops comprising several segments completely in the core store. A significant result related to this is the number of segment places needed to reduce the drum transfer time to a negligible size. This is shown by Figure 2 to be 6 and by Figure 3 to be 10. Allowing for variables in the core store it would seem from this that the segments should be kept so short that at least 20 can be held in the core store.

A lower limit to the segment size imposes itself by the necessity of supplying a code indicating the segment termination at the end of each segment. The fraction of the storage capacity used for this purpose will increase as the segment size is reduced and will eventually drown out all the useful information. This capacity effect is added to the increase of the segment transition time, whose importance depends greatly on the hardware facilities, however.

Although this discussion is inconclusive there is nothing to indicate that a segment size in GIER ALGOL of, say, 2 drum tracks, would offer any advantages over the simple choice actually adopted.

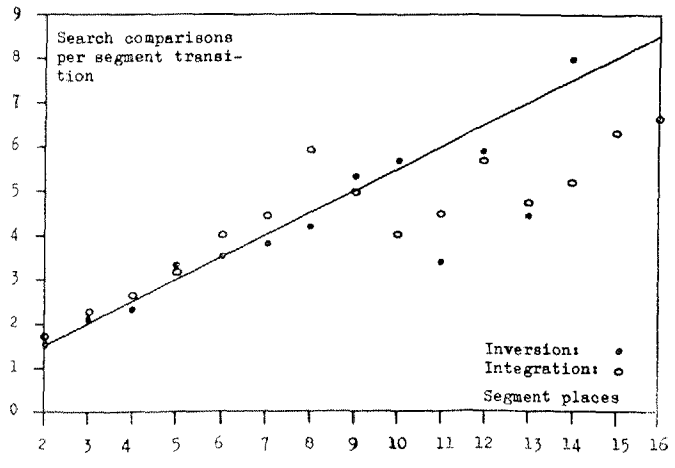

FIG. 4

Conclusions

In view of the fact that the segment administration described above was realized entirely through machine facilities which in no way had been designed with this problem in mind, the performance achieved seems very satisfactory. The approach can therefore be recommended for use in machines of a similar structure. Even better performance may be achieved if one rather special machine instruction is available. This would therefore be an important feature to include in new machines.

The importance of the characteristics of ALGOL 60 for the success of the system may also be noticed. Indeed, the approach depends directly on the block structure of the language and the attendant possibility of using a stack for the variables of the program.

Acknowledgements. The ideas described in the present paper are the outcome of the continued collaboration of Jørn Jensen and the present writer. The remarkably concise and effective machine coding of the fixed administration was done by Jørn Jensen. In discussing the possibility of a special machine instruction for performing the segment transition we have been assisted by Per Mondrup, who did the necessary microprogramming.

APPENDIX 1
Program Storage Administration

The following ALGOL program gives the essentials of the program storage administration held permanently in the core store at run time. It is hoped that the choice of identifiers and the comments given will make further explanations unnecessary.

```
integer MAX SEGMENT, CURRENT PLACE, LAST USED,
    CURRENT PRIORITY;
integer array SEGMENT TRACK, PRIORITY[1: 20];
comment These variables define the state of the system. MAX
    SEGMENT gives the number of segments held in the core store.
    Depending on the number of locations reserved for variables
    this will be an integer between 2 and 20. CURRENT PLACE
    gives the number of the segment place holding the program
    being executed. For simplicity the program assumes that the
    segments are stored consecutively from address 0. LAST USED
    is the smallest address pointing to locations used for variables.
    CURRENT PRIORITY is the priority of the segment being
    executed. SEGMENT TRACK and PRIORITY list the track
    number and associated priorities of the segments presently
    held in the core store. Only the first elements, up to index =
    MAX SEGMENT, are meaningful;
procedure TRANSFER CONTROL(TRACK, RELATIVE AD-
    DRESS);
integer TRACK, RELATIVE ADDRESS;
comment This procedure must be called from the translated
    program every time the control should be transferred to an-
    other segment;
begin integer i, min priority;
for CURRENT PLACE := 1 step 1 until MAX SEGMENT do
```

```
    if TRACK = SEGMENT TRACK[CURRENT PLACE] then
      go to SET PRIORITY;
  comment Track is not in core store;
  min priority := CURRENT PRIORITY;
  for i := 1 step 1 until MAX SEGMENT do
    if PRIORITY[i] < min priority then
    begin min priority := PRIORITY[i]; CURRENT PLACE :=
      i end;
  transfer from drum(TRACK) to address: (CURRENT PLACE ×
    41);
  SEGMENT TRACK[CURRENT PLACE] := TRACK;
  if (MAX SEGMENT + 1) × 41 < LAST USED then
    begin comment Release of a segment;
    MAX SEGMENT := MAX SEGMENT + 1;
    SEGMENT TRACK[MAX SEGMENT] := 0;
    comment The newly released segment is not used at once, in
      order to counteract the ill effects of a rapidly fluctuating stack
      of variables in the core store;
    PRIORITY[MAX SEGMENT] := CURRENT PRIORITY;
    end;
  SET PRIORITY:
  PRIORITY[CURRENT SEGMENT] := CURRENT PRIORITY
    := CURRENT PRIORITY + 1;
  if CURRENT PRIORITY > 511 then
    begin comment Program storage cleanup: rarely used seg-
      ments at high segment numbers are cancelled;
    Boolean cancel; integer p;
    cancel := true;
    for i := MAX SEGMENT step −1 until 1 do
      begin p := PRIORITY[i] ÷ 256;
      if cancel then cancel := p = 0 ∧ i > 2;
      if cancel then MAX SEGMENT := MAX SEGMENT − 1
        else PRIORITY[i] := p;
      end;
    CURRENT PRIORITY := 2
    end clean up;
  go to instruction[41 × CURRENT PLACE + RELATIVE AD-
    DRESS];
  end TRANSFER CONTROL;
```

APPENDIX 2
Test Programs

The full texts of the test programs given below include several calls of standard procedures of the GIER ALGOL III system. The effects of these are given briefly as follows:

drum place. An internal variable whose value controls the part of the drum involved in calls of from drum and gierdrum.
from drum. Transfers values from the drum into an array given as parameter.
gierdrum. Reads binary code from the input tape to the drum.
gierproc. Jumps to the machine code given as parameter.
kbon. A Boolean procedure giving the state of a manual switch.
output. Punches the arithmetic value of the second parameter.
typechar. Waits for input of a character from the typewriter.
typein. Waits for input of a number from the typewriter.
write. Types the arithmetic value of the second parameter.
writecr. Types one carriage return character.
writetext. Types a text.

The Matrix Inversion Program

```
begin
procedure gjr (a, n, eps, singular);
value n, eps; array a; integer n; real eps; label singular;
comment Inversion of matrices by the method of Gauss-Jordan
  [SCHWARZ, H. R. An introduction to ALGOL. Comm. ACM 5
  (Feb. 1962), 82-95]. n is the order of the matrix a[i, k] to be
  inverted. eps is a tolerance for acceptance of the singularity
  of the given matrix and singular is the emergency exit in case
  of a singular matrix;
begin integer i, j, k; real pivot, z; array b, c[1: n];
  integer array p, q[1: n];
for k := 1 step 1 until n do
  begin comment: determination of the pivot element;
  pivot := 0;
  for i := k step 1 until n do for j := k step 1 until n do
    if abs (a[i, j]) > abs (pivot) then
      begin pivot := a[i, j]; p[k] := i; q[k] := j end;
  if abs(pivot) ≤ eps then go to singular;
  comment: exchange of the pivotal row with the kth row;
  i := p[k];
  if i ≠ k then for j := 1 step 1 until n do
    begin z := a[i, j]; a[i, j] := a[k, j]; a[k, j] := z end j;
  comment: exchange of the pivotal column with the kth
    column;
  j := q[k];
  if j ≠ k then for i := 1 step 1 until n do
    begin z := a[i, j]; a[i, j] := a[i, k]; a[i, k] := z end i;
  comment: Jordan step;
  for j := 1 step 1 until n do
    begin
    if j = k then
      begin b[j] := 1/pivot; c[j] := 1 end
    else begin b[j] := −a[k, j]/pivot; c[j] := a[j, k] end;
    a[k, j] := a[j, k] := 0
    end j;
  for i := 1 step 1 until n do
    begin z := c[i];
    for j := 1 step 1 until n do a[i, j] := a[i, j] + z × b[j]
    end for i
  end k;
comment: reordering of the matrix;
for k := n step −1 until 1 do
  begin
  j := p[k];
  if j ≠ k then for i := 1 step 1 until n do
    begin z := a[i, j]; a[i, j] := a[i, k]; a[i, k] := z end i;
  i := q[k];
  if i ≠ k then for j := 1 step 1 until n do
    begin z := a[i, j]; a[i, j] := a[k, j]; a[k, j] := z end j
  end k
end gjr;

integer d, w;
d := drum place; gier drum (33, w);
comment Input of machine code to drum;
  begin boolean array T[1: w]; integer p;
  writetext (⟨<
Sæt KB⟩); typechar; drumplace := d; from drum (T);
comment Machine code to T;
for p := 0 step 1 until 25 do
  begin array A[0: 41×p], a[1: 10, 1: 10]; integer i, j;
if kbon then gierproc(T[3]) else typechar;
for i := 1 step 1 until 10 do
for j := i step 1 until 10 do a[i, j] := a[j, i] := 11 − j;
gjr(a, 10, 10-8, nd);
nd:
if kbon then begin gierproc(T[2]); output (⟨ndd⟩, p) end
    else begin writecr; write (⟨ndd⟩, p); i := typein end;
  end
  end
end;
```

(Continued on page 68