Int. J. Appl. Engng Ed. Vol. 2, No. 1, pp. 61–65, 1986
Printed in Great Britain.

0742–0269/86 $3.00+0.00
Pergamon Journals Ltd.

Encoding of Numbers to Detect Typing Errors*

H. P. GUMM
Fachbereich Mathematik, Gesamthochschule Kassel, Heinrich Plett Straße 40, 3500 *Kassel, F.R.G.*

Typing errors which typically occur when large numbers, e.g. account numbers, personal numbers or parts numbers, are entered into keypads can be detected using a single check digit. Our method detects: (a) all one digit errors; and (b) all transpositions of adjacent digits. With conventional methods at least one type of transposition was undetectable. We show why this is so and why a non-commutative addition on the digits {0, 1, . . . , 9} as provided by the 'dihedral group' is superior to the familiar 'addition modulo 10' for the computation of a check digit. Thus, apart from the practical use of the check digit algorithm, which is condensed in a short PASCAL program in Section 4, the primary aim of this paper is to show that there are very practical reasons for studying finite algebraic structures such as groups.

1. INTRODUCTION

IF A WORD is misspelt within a text, it is usually easy to detect the error and to correct it. The misspelt word is not in our language or may simply be unspeakable like 'SHCOOL'. Such a transposition of two neighbouring digits is a common mistake in machine written text. The keys corresponding to the transposed letters have simply been hit in the wrong order. If, however, the word 'FORM' is encountered, there is no way of telling whether maybe 'FROM' was meant unless we read the whole sentence containing the dubious word. It might well be that actually 'FIRM' or 'FARM' was meant, but accidentally the wrong key was hit, another common typing mistake. Again, the context will give us a clue to the intended word. If a bank clerk makes such a typing mistake while entering the account number for some transaction, there is at first glance no way of detecting a typing mistake and one can imagine the trouble if it goes unnoticed.

A 'context' to the account number could be provided by the owners name, so anytime the account number is referred to, that name has to be added. But if the name is as common as, say, SMITH, a mixup might still occur.

There is too much redundancy in this method and it is not used efficiently enough. More seriously, if money is transferred to another bank, a typing error would only be detected when the invoice reaches the second bank, for only there a list of account holders, paired with their correct account numbers is available.

For these reasons one assigns a *check digit* to the original unsecured number and the original number together with the check digit then becomes the valid account number.

* Paper submitted 3 September 1985. Revised version accepted 14 October 1985.

A little example will show how such a scheme can work in principle. Suppose that customer #4813 opens an account at a bank. Instead of receiving the account number 4813 the bank computes a check digit p so that $4 + 8 + 1 + 3 + p = 0 \pmod{10}$, that is $p = 4$. Now the account number will be 48134 once and for all. If a mistake occurs when at some later transaction the same number is entered into the bank's computer, say 43134 is erroneously entered, that mistake is immediately detected, since $4 + 3 + 1 + 3 + 4 \neq 0 \pmod{10}$. It is not intended to automatically correct the erroneous number, the sole purpose is to request the typist for a new input of the correct number.

If typing errors would occur randomly, the above scheme would be optimal, since exactly 90% of all typing errors would be detected. There are, however, typical typing errors, as indicated in the introductory examples. Empirical studies [1, 2] have shown that the following three types of mistakes occur most frequently when data is entered into keypads.

(a) *Single digit errors*: one digit is mistyped because either the wrong key is hit or a digit was misread
example:

$$4711 \rightarrow 4911.$$

(b) *Format errors*: one or more digits are erroneously inserted or left out
example:

$$4711 \rightarrow 43711$$

or

$$4711 \rightarrow 471.$$

(c) *Transposition errors*: two neighbouring digits are switched. The key for the second digit is touched before the key for the first digit was hit. This mistake can also have linguistic reasons, since in German, e.g. two-digit numbers are pronounced 'backwards' (72 is read 'two-and-seventy')

example:

$$4711 \to 7411.$$

There are too many possibilities for format errors to be detectable by a single check digit. But by requiring (and checking) that all numbers have the same format (e.g. account numbers all take eight digits), they can easily be taken care of. We thus concentrate on the two other types of errors, *single-digit errors* and *transposition errors.*

2. STANDARD METHODS AND THEIR LIMITATIONS

We now scan a number of methods for assigning a check digit to a given number and see how the above mentioned mistakes are being taken care of. We always assume that the number n has the digits $d_k, d_{k-1}, \ldots, d_2, d_1$ so that $n = \Sigma d_i 10^{i-1}$. Let p be the check digit, then the new number will read $d_k, d_{k-1}, \ldots, d_2, d_1, p$.

2.1. *Parity check*

We compute p so that

$$d_k + d_{k-1} + \ldots + d_2 + d_1 + p = 0 \pmod{10}.$$

Example: original number:

$n = 9\,0\,1\,4\,8\,3\,2$

$p = 10 - (9 + 0 + 1 + 4 + 8 + 3 + 2) \pmod{10}$

$= 3$

encoded number: 9 0 1 4 8 3 2 <u>3</u>.

An encoded number is correct, if and only if the sum of its digits is 0 (mod 10). Clearly, one-digit errors will all be detected, but transposition errors will never be found since $\ldots d_i + d_{i-1} \ldots = \ldots d_{i-1} + d_i \ldots$ as a consequence of the commutative law for addition modulo 10.

2.2. *Weighted parity check*

We impose a weight w_i on every digit position. Each digit is first multiplied with the weight corresponding to its position and then added. Let the sequence of weights be $(w_0, w_1, w_2, \ldots)$ then we choose p so that

$$w_k d_k + w_{k-1} d_{k-1} + \ldots + w_2 d_2 + w_1 d_1 + w_0 p = 0 \pmod{10}.$$

Example: with the weights $(w_0, w_1, w_2, \ldots) = (1, 3, 7, 9, 1, 3, 7, \ldots)$ and the original number $n = 9\,0\,1\,4\,8\,3\,2\,3$ we find

9	0	1	4	8	3	2	3	p	digits
1	9	7	3	1	9	7	3	1	weights

$$9 + 0 + 7 + 12 + 8 + 27 + 14 + 9 + p = 6 + p \pmod{10}.$$

In order for this to add up to 0 (mod 10) we find $p = 4$, so the encoded number becomes 9 0 1 4 832 3 4. An encoded number is considered correct, if its weighted sum is equal to 0 (mod 10).

With the weights in the example, all single digit errors will be detected, but notice that a transposition error like . . . 8 3 . . . → . . . 3 8 . . . cannot be detected.

At first glance one might want to choose a different sequence of weights, but it is immediately clear, that 1, 3, 7, 9 are the only possible weights if single digit errors are to be detected. w_i cannot be even, because then $w_i \cdot d_i = w_i d_i'$ whenever $(d_i - d_i') = 5 \pmod{10}$ so a mistake changing d_i into d_i' would not be detected. With a similar argument, w_i cannot be 5. (Mathematically speaking, w_i must be relatively prime to 10.) Turning to transposition errors $\ldots d_i d_{i-1} \ldots \to \ldots d_{i-1} d_i \ldots$ we see that these errors cannot be detected if $(d_i - d_{i-1}) = 5 \pmod{10}$. The reason is that $w_i - w_{i-1}$ is even, so

$$(w_i - w_{i-1})(d_i - d_{i-1}) = (w_i - w_{i-1}) \cdot 5 = 0 \pmod{10},$$

so

$$\ldots w_i d_i + w_{i-1} d_{i-1} \ldots = \ldots w_i d_{i-1} + w_{i-1} d_i \ldots.$$

In spite of its shortcomings, the weighted parity check was an improvement over the simple parity check, since it detects all single digit errors and all those transposition errors where $(d_i - d_{i-1}) \neq 5 \pmod{10}$. The key was, to transform a digit x occurring at position i into $w_i x$. The generalization is obvious now, we choose transformations τ_i for every digit position, where a transformation is simply a map from $D = \{0, 1, \ldots, 9\}$ to itself.

So a *transformation method modulo 10* would consist of a sequence of transformations τ_i such that the check digit p is computed so that

$$\tau_k(d_k) + \ldots + \tau_1(d_1) + \tau_0(p) = 0 \pmod{10}.$$

Usually one would choose $\tau_0 = id$, the identity, so this equation can be trivially solved for p. Clearly for every $x \neq y$ and every position i we need $\tau_i(x) \neq \tau_i(y)$, since otherwise a single digit error at position i, changing x to y or vice versa, would be undetectable. Thus each τ_i has to be one to one and therefore a permutation of the digits $\{0, \ldots, 9\}$.

2.3. *The EKONS system*

As an example of such a transformation system we consider the EKONS system which is used by the West German 'Sparkassen' (Savings banks). The transformations used are $\tau_i =$ identity if i is even and $\tau_i(x) = \tau(x) =$ checksum $(2x)$ when i is odd.

Example: (EKONS system)

4	9	0	7	8	6	2	3	p
$\downarrow id$	$\downarrow \tau$	$\downarrow id$	$\downarrow \tau$	$\downarrow id$	$\downarrow \tau$	$\downarrow id$	$\downarrow \tau$	$\downarrow id$

$$4 + 9 + 0 + 5 + 8 + 3 + 2 + 6 + p = 7 + p \quad \pmod{10}$$

In order to add to 0 (mod 10), $p = 3$, so the encoded number is 4 9 0 7 8 6 2 3 3.

Since the transformations used are permutations, we know that single digit errors will be detected. What about transposition errors? It is easy to see that almost all transposition errors will be detected, except for the error . . . 09 . . . → . . . 90 . . . . If digits a and b appear in neighbouring places in the number . . . ab . . . with, say, a at an odd numbered place, then the contribution to the final sum is $\tau(a) + id(b) = \tau(a) + b$. If a and b are transposed their contribution is $\tau(b) + a$, instead. This is not detected when $\tau(a) + b = \tau(b) + a$, i.e. $\tau(a) - a = \tau(b) - b$. With $\tau(x) =$ checksum $(2x)$ we find that $\tau(9) - 9 = \tau(0) - 0$ is the only solution with $a \neq b$.

So the question remains whether the transformations can be chosen in some clever way so that actually all transposition errors are detected. The following proposition shows that no transformation method modulo 10 can perform better than the EKONS system.

2.4. *Proposition*

Every transformation method modulo 10 leaves some transposition error . . . ab . . . → . . . ba . . . undetected.

Proof. If a transposition . . . ab . . . → . . . ba . . . at positions i, $i + 1$ is to be noticed, we need

$$\tau_{i+1}(a) + \tau_i(b) \neq \tau_{i+1}(b) + \tau_i(a) \ (\text{mod } 10),$$

thus

$$\tau_{i+1}(a) - \tau_i(a) \neq \tau_{i+1}(b) - \tau_i(b) \ (\text{mod } 10).$$

Denoting the transformation $x \mapsto \tau_{i+1}(x) - \tau_i(x)$ by τ, we find that $\tau(a) \neq \tau(b)$ (mod 10). Requiring this for all $a \neq b$ we conclude that τ must be a permutation of the digits $\{0, \ldots, 9\}$. The following Lemma shows that a τ defined as above can never be a permutation, thus finishing the proof.

2.5. *Lemma*

If α and β are permutations of the digits $\{0, \ldots, 9\}$ then their difference, $\tau = \alpha - \beta$ (mod 10) is never a permutation.

Proof. Since α and β are one to one, we have

$$D = \{0, \ldots, 9\} = \{\alpha(x) \mid x \in D\} = \{\beta(x) \mid x \in D\}.$$

If τ is a permutation the same holds for τ, so

$$D = \{\tau(x) \mid x \in D\} = \{\alpha(x) - \beta(x) \mid x \in D\}.$$

But taking the sum (mod 10) over all elements of the above sets we obtain the contradiction

$$5 = \sum_{x \in D} x = \sum_{x \in D} \tau(x) = \sum_{x \in D} \alpha(x) - \beta(x)$$

$$= \sum_{x \in D} \alpha(x) - \sum_{x \in D} \beta(x) = 0.$$

3. A DIFFERENT KIND OF ADDITION

After the negative results of the last section it is clear now that in order to construct a check digit method detecting all single digit errors as well as all transposition errors we have to modify more than only the digit transformations. We will also have to exchange the 'addition modulo 10' for a different kind of operation, call it □. With such a new 'addition' we plan to compute the check digit as before to be the number p with

$$\tau_k(d_k) \,\square\, \tau_{k-1}(d_{k-1}) \,\square\, \ldots \,\square\, \tau_1(d_1) \,\square\, \tau_0(p) = 0. \qquad (1)$$

Which requirements do τ_i and which does □ have to satisfy?

For the τ_i this is easily answered as before, they have to be one to one and therefore permutations of the digits $\{0, \ldots, 9\}$. Suppose now that $a \,\square\, b = a \,\square\, c$ for some digits $a, b, c \in \{0, \ldots, 9\}$. Then, given a position i, we let $a' = \tau_i^{-1}(a)$, $b' = \tau_{i-1}^{-1}(b)$, $c' = \tau_{i-1}^{-1}(c)$, so

$$\tau_i(a') \,\square\, \tau_{i-1}(b') = \tau_i(a') \,\square\, \tau_{i-1}(c')$$

which means, that an error, changing b' to c' would not be detected. Thus we must have $b' = c'$, hence $b = c$. So the first requirement for □ is

$$a \,\square\, b = a \,\square\, c \Rightarrow b = c \qquad (2)$$

and symmetrically

$$b \,\square\, a = c \,\square\, a \Rightarrow b = c. \qquad (3)$$

If we write down the operation □ in a 10 × 10 table, where the entry in the ith row and the jth column is the product $i \,\square\, j$, then the above requirements mean: (a) the entries of every row are mutually different; and (b) the entries of every column are mutually different. Such a table, or the operation it represents is called a 'latin square'.

There is a large supply of 10 × 10 latin squares, yet to be able to 'add' more than two numbers without specifying the order in which this has to be done we require that the associative law should hold

$$a \,\square\, (b \,\square\, c) = (a \,\square\, b) \,\square\, c. \qquad (4)$$

This law enables us to leave out brackets when 'adding' a large expression like eqn (1).

Equations (2)–(4) taken together imply that □ is a group operation, thus there must be a neutral element 0 with $0 \,\square\, x = x \,\square\, 0 = x$ for all x and an inverse x^- to every $x \in \{0, \ldots, 9\}$ such that $x^- \,\square\, x = x \,\square\, x^- = 0$. It is well known that there exist exactly two groups on $\{0, \ldots, 9\}$. The first is given by addition modulo 10 and the second is the so-called 'dihedral group' (Fig. 1). Note that the operation is not commutative, e.g. $3 \,\square\, 6 \neq 6 \,\square\, 3$. For the detection of transposition errors though, this should not be a disadvantage!

Now that 'addition' is fixed, let us turn to the

□	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	0	6	7	8	9	5
2	2	3	4	0	1	7	8	9	5	6
3	3	4	0	1	2	8	9	5	6	7
4	4	0	1	2	3	9	5	6	7	8
5	5	9	8	7	6	0	4	3	2	1
6	6	5	9	8	7	1	0	4	3	2
7	7	6	5	9	8	2	1	0	4	3
8	8	7	6	5	9	3	2	1	0	4
9	9	8	7	6	5	4	3	2	1	0

Fig. 1. The 'addition table' for the dihedral group.

permutations τ_i that are needed. We have to fix them so that transposition errors are also detected. This means, that for all a, b and every digit position i we have

$$\tau_i a \,\square\, \tau_{i-1} b = \tau_i b \,\square\, \tau_{i-1} a \Rightarrow a = b. \qquad (5)$$

We set $u := \tau_{i-1} a$, $v := \tau_{i-1} b$ and $\tau := \tau_i \tau_{i-1}^{-1}$, so

$$\tau u \,\square\, v = \tau v \,\square\, u \Rightarrow u = v. \qquad (6)$$

Thus detection of transposition errors is guaranteed if we can find a sequence τ_i of permutations such that each $\tau := \tau_i \tau_{i-1}^{-1}$ satisfies eqn (6).

On the other hand, once we have found one permutation satisfying eqn (6), then we may simply set $\tau_0 = id$ and $\tau_i := \tau^i = \tau \circ \tau_{i-1}$, since then $\tau_i \circ \tau_{i-1}^{-1} = \tau^i \circ (\tau^{i-1})^{-1} = \tau^{i-(i-1)} = \tau$, which satisfies eqn (6).

Let us collect what we know.

3.1. *Theorem*

Let $\square$ be the operation of the dihedral group on the digits $D = \{0, \ldots, 9\}$ and τ a permutation on D satisfying $\tau u \,\square\, v = \tau v \,\square\, u \Rightarrow u = v$, then choosing p so that

$$\tau^n d_n \,\square\, \tau^{n-1} d_{n-1} \,\square\, \ldots \,\square\, \tau^2 d_2 \,\square\, \tau d_1 \,\square\, p = 0$$

yields a check digit method detecting all single digit errors and all transposition errors.

Note that nothing special about $\square$ was used so far. The theorem is also true (but worthless) if we exchange $\square$ for $+$, since Lemma 2.5 implies that an appropriate τ for $+$ does not exist.

In Ref. [3] we showed that check digit methods exist for arbitrary number systems, except in base 2, so τ and $\square$ according to eqn (6) had to be constructed abstractly without reference to the number system. Since here we are only concerned with decimal numbers, it suffices to present one such τ concretely and show that it has the desired property.

Proposition. Let τ be the permutation on D with cycle representation (14) (23) (58697), then τ satisfies eqn (6).

Proof. For every pair $u, v \in D$ this can of course be easily checked; to see the claim at a glance, we write down the table for $\tau(u) \,\square\, v$ in matrix form. It simply arises from the table for $\square$ by applying the permutation τ to the rows of $\square$. Now this new table has entry $\tau(x) \,\square\, y$ in position (x, y). One sees at a glance that the entries in position (x, y) are always different from the entries in position (y, x), unless $x = y$, so $\tau x \,\square\, y \neq \tau y \,\square\, x$ unless $x = y$.

Summing up we obtain

Theorem. Let τ be the permutation (14) (23) (58697) on the digits $D = \{0, \ldots, 9\}$ and let $\square$ be the operation of the dihedral group on D as given in Fig. 1. Computing the check digit p for a number having digits $d_k, \ldots, d_1$ as $p = [\tau^k d_k \,\square\, \ldots \,\square\, \tau d_1]^-$ yields a check digit detecting *all single digit errors* and *all errors arising from transposition of adjacent digits*. (Here $[\ldots]^-$ denotes the inverse with respect to $\square$.)

Note finally that τ has been chosen so that $\tau(0) = 0$. Thus, when computing a check digit for a number, leading zeros do not change the check digit.

4. IMPLEMENTATION

The implementation of the described method turns out to be extremely simple and efficient. For aesthetic reasons we introduce types for digits and

```
type  digit     = 0 .. 9;
      position  = 0 .. maxlength;
      bignumber = array [0 .. maxlength] of digit;

function add(x,y:digit):digit;
begin
  if x < 5 then
                begin
                  if y < 5 then add :=  (x+y) mod 5
                           else add := ((x+y) mod 5) + 5
                end
           else
                begin
                  if y < 5 then add := ((x-y) mod 5) + 5
                           else add := (x-y+5) mod 5
                end
end;

function inv(x:digit):digit;
begin
  if x < 5 then inv := (5-x) mod 5
           else inv := x
end;

function tau(i:position;x:digit):digit;
begin
  if x = 0 then tau := 0
           else if x < 5 then if ((i mod 2) = 0)
                                then tau := x
                                else tau := 5-x
                         else tau := ((3*i + x) mod 5) + 5
end;
```

Fig. 2. Types and auxiliary functions.

```
function checkdigit(number:bignumber):digit;
var  dig,sum,aux : digit;
                 k : position;

begin
  sum := 0;                              (* initialize           *)
  for k := 1 to maxlength do             (* for all digits ..    *)
     begin
        dig  := number[k];               (* take next digit      *)
        aux  := tau(k,dig);              (* apply tau-to-the-k   *)
        sum  := add(aux,sum);            (* add up               *)
     end;
  checkdigit := inv(sum);                (* take inverse         *)
end;

function correct(number:bignumber):boolean;
var  dig,sum,aux : digit;
                 k : position;

begin
  sum := 0;
  for k := 0 to maxlength do             (* for all digits in-   *)
                                         (* cluding checkdigit   *)
     begin
        dig  := number[k];               (*  same                *)
        aux  := tau(k,dig);              (*       as             *)
        sum  := add(aux,sum);            (*           before     *)
     end;
  correct   := (sum = 0);                (* if sum = 0 then o.k. *)
end;
```

Fig. 3. The main functions *checkdigit* and *correct*.

000	011	022	033	044	058	069	075	086	097
104	110	121	132	143	159	165	176	187	198
203	214	220	231	242	255	266	277	288	299
302	313	324	330	341	356	367	378	389	395
401	412	423	434	440	457	468	479	485	496
506	517	528	539	545	552	563	574	580	591
607	618	629	635	646	651	662	673	684	690
708	719	725	736	747	750	761	772	783	794
809	815	826	837	848	854	860	871	882	893
905	916	927	938	949	953	964	970	981	992

Fig. 4. The first 100 secured numbers.

digit positions. Maxlength is initialized as a constant, describing the maximum number of digits an unsecured number may have. Since the numbers could exceed the largest machine number we store them as 'bignumbers'.

The auxiliary functions *add* (for □), *tau* (for τ) and *inv* (for $^{-}$) could simply be tabulated as matrices and initialized with a DATA statement by a FORTRAN programmer (Fig. 2). Note that *tau* has become binary, tau(i, x) is $\tau^i(x)$, which certainly saves a lot of computational steps. Furthermore note from the definition of τ that $\tau^{10} = id$, hence tau($i + 10, x$) = tau(i, x), which is important if tau should be tabulated.

The functions we are interested in are *checkdigit* and *correct* (Fig. 3). *Checkdigit* computes a checkdigit for a bignumber and *correct* checks a bignumber for correctness. Note the similarity between both functions.

Figure 4 displays a list of the first 100 numbers together with their correct checkdigit. Try it out and check that transposing any two digits will never yield a number in the table again. The same holds for changing any digit to a different one.

REFERENCES

1. H. P. Gumm, A new class of check-digit methods for arbitrary number systems. *IEEE Trans. Inf. Theory* **31**, 102–105 (1985).
2. W. Kunerth, *EDV-gerechte Verschlüsselung: Grundlagen und Anwendung moderner Nummernsysteme*. Forkel Verlag, Stuttgart (1981).
3. H. Schechinger, Kontrolle mit Hilfe von Prüfziffern, *Bürotechnische Sammlung, BTS-systematisch*, Heft 171. März (1969).