

1-2/2018

Markos Mitsos

**Matching under inequality
via sorting algorithms**

Ralf Herminghaus

**KALHA - Erfahrungen bei
der Implementation von
neuronalen Netzen in APL**

Dieter Kilsch

APL2:  kann C

IM BLICKPUNKT

Ralf Herminghaus

Programmieraufgabe



Liebe Mitglieder von APL Germany,
liebe APL-Freunde,

Sie lesen das neue *APL-Journal*. Es beinhaltet drei Beiträge, die auf Vorträgen der Autoren auf den Tagungen der letzten beiden Jahre basieren. Ich wünsche Ihnen viel Freude bei der Lektüre dieser Ausgabe.

In den beiden vergangenen Jahren gab es im Verein einige Veränderungen. Ich habe vor zwei Jahren den Vorsitz übernommen und zu Beginn den Internetauftritt verbessert. Hier sollen alle Tagungen erwähnt und wenn möglich auf die Vorträge verwiesen werden. Im vergangenen Jahr musste die DSGVO umgesetzt werden. Per Rundschreiben haben wir informiert und die Erlaubnis zum Verarbeiten der Daten eingeholt.

Inzwischen sind wir einen Schritt weiter gegangen und haben im vergangenen Herbst mit der anstehenden Neufassung der Satzung den Datenschutz in der Satzung verankert. Diese Änderung ist auf dem Weg zum Amtsgericht und wird vermutlich bis zur nächsten Mitgliederversammlung im Mai rechtskräftig.

Über die finanzielle Lage des Vereins wurde im Vorstand und auf der Mitgliederversammlung im November intensiv diskutiert. Die zurückgehenden Mitgliederzahlen reduzieren die Einnahmen: die Ausgaben, vor allem für das Journal, reduzieren die Rücklagen deutlich. Die Mitgliederversammlung hat daher den Übergang zur Veröffentlichung des Journals in elektronischer Form auf unserer Internetseite beschlossen. Die Mitglieder werden per E-Mail über Neuerscheinungen des Journals informiert. Als Alternative stand die Erhöhung der Mitgliederbeiträge im Raum, die durch die beschlossenen Maßnahmen jetzt jedoch konstant gehalten werden können.

Ein weiterer wichtiger Punkt meiner Arbeit als Vorsitzender ist der Kontakt zu benachbarten APL-Vereinigungen. Ich habe die Schweizer Gruppe um Urs Oswald besucht und beabsichtige auch an der Tagung der British APL Association teilzunehmen, die für Mai von Paul Grosevenor angekündigt ist.

Unsere nächste Tagung veranstalten wir zusammen mit der GSE-Tagung am 13./14.Mai in Stuttgart. Die Mitgliederversammlung findet am 14. Mai statt. Ich hoffe auf viele Vortragsangebote und eine rege Teilnahme.

Dieter Kilsch

INHALT

Matching under inequality
via sorting algorithms

3

KALAHA - Erfahrungen bei der
Implementation von neuronalen Netzen in APL

39

APL2:  kann C

52

Eine Programmieraufgabe

57

Impressum

62

Matching under inequality via sorting algorithms

Dr. Markos Mitsos*
 Deutsche Krankenversicherung AG DKV - ERGO
 Actuarial Department

January 8, 2017

Abstract

The solution of many problems encountered while processing data with APL depends on matching algorithms. This is taken to mean that data points are searched in a reference list; the found matches are noted and used subsequently. In the case of vectors basic APL provides the necessary primitive to conduct the search via Index Of as in

```
IND ← REF ι DAT.
```

The more general case where (rows of) matrices are to be matched can be tackled through enclosing rows.¹ This solution though leads to very slow processing. Instead a sorting algorithm can be used. (In fact *some* algorithm underlies ι itself as well...)

This paper discusses the generalization of the problem to different kinds of matching under inequality. In one case a possible solution with the help of “special sorting” algorithm is proposed. For the latter the APL primitive Grade Up \uparrow cannot be used.

A big part of the article discusses simple examples or provides the motivation to examine matching problems in a more general setting. This parts could be used in a presentation and may of course be skipped by the expert. We try to use APL code which is basically dialect independent but uses control structures for legibility.

Contents

1 Matching under equality via sorting algorithms in APL	5
1.1 The problem of matching rows of matrices	5
1.2 Solutions via APL primitives	6
1.3 Solution with sorting algorithm	6
1.4 Match on equality as a example for sorting algorithms	7

*markos.mitsos@ergo.de

¹In newer version of Dyalog APL however the primitive ι has been enhanced to encompass the case of matrices.

2	Matching with “final” inequality	7
2.1	Cases where a match is not an equality	7
2.2	A sorting algorithm as solution	8
2.3	Strict inequality	9
2.4	Other relations	9
3	Matching with “initial” inequality	10
3.1	Cases with initial inequality	10
3.2	The first type of match on initial inequality	11
3.3	A different type of match on initial inequality	12
3.4	A practically not very important match type	13
3.5	A match problematic on the APL side	15
3.6	Some remarks on matches and their different types	15
4	Matching on set equality and relation	15
4.1	Setting	15
4.2	Matches in general	16
4.3	Definition of some interesting types of match	16
4.4	Analysis of the different types of match	18
4.4.1	Comparison and true inclusions	18
4.4.2	False inclusions	19
4.4.3	“Commutativity” of relations	19
4.5	Disjoint dissections	20
5	Algorithms for two kinds of match	20
5.1	A sorting algorithm for strong local matches	20
5.1.1	Description	20
5.1.2	Algorithm	21
5.1.3	Proof of correctness	22
5.2	A sorting algorithm for strong global matches	23
5.2.1	Introduction	23
5.2.2	Algorithm	23
5.2.3	Proof of correctness	25
5.2.4	An alternative approach	25
6	Tentative algorithms for problematic matches	26
6.1	A tentative sorting algorithm for weak local matches	26
6.1.1	Introduction	26
6.1.2	Algorithm with a gap	26
6.2	Correctness of algorithm	28
6.2.1	Prerequisites for correctness	28
6.2.2	Proof of correctness	29
6.3	Special sort algorithm with scaffold	30
6.3.1	Basic facts about sort algorithms	30
6.3.2	A false special sort algorithm	31
6.3.3	A modification of (Natural) Merge Sort as special sort	31
6.3.4	Modified proof of correctness	32
6.4	Implementation of the special sort with scaffold	33
6.4.1	Special sort in APL	33
6.4.2	Special sort in .Net	35

6.5	A sorting algorithm for weak global matches	36
6.5.1	Outline of a possible algorithm	36
6.5.2	Outline of implementation of trivial case in APL	37
6.5.3	Discussion of the problem	37
6.5.4	A possibly suboptimal circumvention	38

List of Algorithms

1.4	match on equality	7
2.2	match on final inequality \leq	8
2.3	match on final inequality $<$	9
2.4	match on final inequality \geq	9
2.4	match on final inequality $>$	9
3.2	example of match on initial inequality	11
3.4	example of match on two \leq -inequalities	13
5.1	strong local match	21
5.2	strong global match	23
6.1	weak local match	26
6.4	special sort in APL	33
6.4	special sort in .Net	33

1 Matching under equality via sorting algorithms in APL

This (first) section is an introduction. The presented problems and algorithms are only intended as examples and a motivation for more complex cases.

1.1 The problem of matching rows of matrices

We want to match data, denoted by DAT, with a reference list, denoted by REF, in APL. We are interested in the case of matrices with the same number of columns (possibly one) and will always assume that DAT and REF are such.

We want to find the first occurrence of each row of DAT in REF (meaning the first identical row). We are only interested in an effective and fast method to solve the problem in case at least one of the matrices is big (at the very least some tens of thousands of rows) and the other at least medium sized (at the very least some hundreds of rows).

That fore we ignore the general case of arbitrary depth and assume the matrices are simple. We can then further assume that they are numeric (for example by replacing alphanumerical data with their Unicode code).

In fact in most cases we will only be interested in integers. Of course the discussed algorithms can be generalized to other data types.

1.2 Solutions via APL primitives

There are of course many possible solutions to the problem using APL primitives. For example the “monster comparison”

$$\text{IND} \leftarrow (\iota 1 \uparrow \rho\text{REF}) \times [2] \text{DAT} \wedge. = \Phi\text{REF} \quad (1)$$

returns all matches and can be used to find the first one. It is of course not of much practical use.

Just the same enclosing the matrices along the second axis will deliver the desired result via

$$\text{IND} \leftarrow (\text{c}[2] \text{REF}) \iota (\text{c}[2] \text{DAT}) . \quad (2)$$

However enclosing big matrices is not very effective and/or fast (although the process has been constantly improved in subsequent versions of many APL dialects). Furthermore the used primitive Index Of may not (depending on the used APL dialect) be optimized for this special kind of data.

Newer versions of Dyalog APL are a special case. Here the APL primitive ι has been enhanced to encompass the case of matrices. As far as publicly known this is done by hashing each row and by this means reducing the problem to ordinary numerical vectors.

1.3 Solution with sorting algorithm

The problem with expression (1) is that it compares all provided pairs of rows (from the data and the reference respectively). Effective ways to compare only “necessary” pairs are encoded in the well known sort algorithms. The APL primitive \uparrow (of course!) uses those.

A possible solution to the problem can be created along the following lines.

1. Catenate the two matrices while keeping track of the origin of each row.
2. Sort the result.
3. Split the result into (disjoint) groups of identical rows.
4. Choose the first element of REF from each group as a match for all elements of DAT in the same group.

The main part of the work is the sort. The principle of sorting and then grouping can also be used to solve many other problems. The algorithm should not depend on the sort being stable.² To achieve this some additional information (like the row numbers of the two matrices) may be used.

²In most APL dialects the Grade Up primitive \uparrow seems to implement a stable sort. There does not seem however to exist a direct warranty for this in the language references! On the other hand the APL Standard demands stable sorting, so any APL dialect adhering to the Standard must provide it.

1.4 Match on equality as a example for sorting algorithms

We write down one possible implementation of the algorithm outlined. A similar function that also takes care of cases a little more general works very well with matrices with significantly more than one million rows.

The algorithm selects the first row of each group (of rows). It must therefore ensure that this is “the right one”, namely the *first* row of REF fulfilling the necessary criteria. (In absence of such a match a dummy must be used.) This is achieved through the selected sort.

The version we give is not necessarily always the fastest but is easy to read. The construction of the sort vector takes up up to 75% of the total runtime (for random data) and is build in a way that does not assume that \uparrow is stable.

Algorithm “match on equality”

```

IND ← (1 ↑ ρDAT) ρ (NO_MATCH ← 1 + 1 ↑ ρREF)
    A result is 4 byte integer

MAT ← REF ,[1] DAT A concatenate
ROW ← ι 1 ↑ ρMAT    A member numbers

V ←  $\uparrow$  MAT , ROW A sorting vector
MAT ← MAT[V;]    A sort matrix
ROW ← ROW[V ]    A sort member numbers

BEG_I ← 1 , v / (1 ↓ [1] MAT) ≠ (−1 ↓ [1] MAT) A inequality?

DAT_I ← NO_MATCH ≤ ROW A mark data

V ← BEG_I / ι 1 ↑ ρMAT    A begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ρMAT)) - V A members per group

IND[(1 - NO_MATCH) + DAT_I / ROW]
    ← DAT_I / MAT_NR / NO_MATCH [ BEG_I / ROW
    A member number of first match in reference

```

2 Matching with one, “final” inequality via sorting algorithms in APL

This section presents a problem that cannot be solved through enclosing (as in expression (2) of section 1). The sorting algorithm “match on equality” of subsection 1.4 however can be modified and used for this purpose. The necessary sort is done by the Grade Up APL primitive \uparrow .

2.1 Cases where a match is not an equality

In many cases the desired match cannot be expected to be an exact one. The most common example in German health insurance is “begin business time”. Let us consider premiums or annuity values for a certain plan.

The plan has been recalculated on certain dates $r_i \in \mathcal{R}$, $i \in \{1, \dots, l\}$. Each person j , $j \in \{1, \dots, m\}$ in this plan has contracted at a certain date d_j . At the time of contract the valid annuities were those with the highest date before or at contract.

The equality is possible but not probable. For each person j the subset

$$\mathcal{R}(d_j) = \{r = r_{max}(d_j)\} \subset \mathcal{R}, \quad r_{max}(d_j) = \max\{r; r \leq d_j\},$$

defines the desired matches.

In this section we will consider problems with a series of matches on equality followed by one last (and therefore “final”) match on inequality, more specific on the condition “less or equal”. For example those could be one or more criteria defining a plan (like plan name and sex) followed by business time.

On the APL side the first columns of our matrices are to be matched on equality (=) whereas the last one is to be matched on “less or equal” (\leq).

2.2 A sorting algorithm as solution

We can still use a (lexicographic) sort to find the desired match. Sorting the leading columns and grouping them is obviously necessary. Sorting additionally the last column brings the data in the right succession.

The grouping however from subsection 1.4 has to be modified. Different dates belong to separate groups exactly when they come from the reference list. All subsequent data point dates with the same leading columns and up to the next reference date belong to the same group regardless of their value.

In our algorithm “match on equality” we just replace

```
BEG_I ← 1 , v / (1 ↓[1] MAT) ≠ (¯1 ↓[1] MAT) ⌘ inequality?
```

and obtain the following.

Algorithm “match on final inequality \leq ”

```
...
V ← ⍋ MAT , ROW ⌘ sorting vector
...
BEG_I ← 1 , v / (1 ↓[1] V) ≠ (¯1 ↓[1] (V ← ¯1 ↓[2] MAT))
    ⌘ inequality in leading columns?
I ← (1 ↓ V) ≠ (¯1 ↓ (V ← , ¯1 ↑[2] MAT))
    ⌘ inequality in final column?
REF_I ← NO_MATCH > ROW
    ⌘ mark references
BEG_I ← BEG_I v (1 , I ~ 1 ↓ REF_I)
    ⌘ final inequalities count at reference only
...
```

The sort is unchanged and only given for better comparison with the following subsections.

2.3 Strict inequality

In some cases a strict inequality may be needed. To obtain a match where the last property (column) is defined by a subset like

$$\mathcal{R}_j = \{r_i = r_{max}(d_j)\} \subset \mathcal{R}, \quad r_{max}(d_j) = \max\{r_i; r_i < d_j\},$$

the same technique may be used.

The only necessary change to the algorithm "match on equality" additionally to the one used to obtain the algorithm "match on final inequality \leq " are

- to replace

```
V ← ⚡ MAT , ROW ⚡ sorting vector
```

with a sort that puts elements of the reference *after* those of the data if all columns are identical and

- to recognize a reference preceded by an identical data point as a group begin.

One possible (though not the most effective) solution is the following.

Algorithm "match on final inequality $<$ "

```
...
V ← ⚡ MAT , (NO_MATCH > ROW) , [1.5] ROW ⚡ sorting vector
...
BEG_I ← 1 , √ / (1 ↓ [1] V) ≠ (¬1 ↓ [1] (V ← ¬1 ↓ [2] MAT))
    ⚡ inequality in leading columns?
I ← (1 ↓ V) ≠ (¬1 ↓ (V ← , ¬1 ↑ [2] MAT))
    ⚡ inequality in final column?
REF_I ← NO_MATCH > ROW ⚡ mark references
BEG_I ← BEG_I √ (1 , (I ~ 1 ↓ REF_I)
    √ ((1 ↓ REF_I) ~ (¬1 ↓ ~ REF_I)))
    ⚡ final inequalities count at reference only,
    final equalities if after data and at reference
...

```

We emphasize that the algorithm works in that form only for the special case of one, final inequality discussed here.

2.4 Other relations

To complete the discussion on matching with one, final inequality we work out the conditions "greater or equal" and "greater". Those problems are complements to the ones discussed in subsections 2.2 and 2.3.

They can be tracked either by reversing the sort (using sort down through Ψ) or by using the *last* element of each group. We use the first option to get algorithms that are as easily comparable as possible.

We obtain the following by replacing the sort in the algorithm "match on final inequality \leq ".

Algorithm “match on final inequality \geq ”

```

...
V ← Ψ MAT , (-ROW) ⌘ sorting vector
...
BEG_I ← 1 , v / (1 ↓[1] V) ≠ (¬1 ↓[1] (V ← ¬1 ↓[2] MAT))
    ⌘ inequality in leading columns?
I ← (1 ↓ V) ≠ (¬1 ↓ (V ← , ¬1 ↑[2] MAT))
    ⌘ inequality in final column?
REF_I ← NO_MATCH > ROW
    ⌘ mark references
BEG_I ← BEG_I ∨ (1 , I ~ 1 ↓ REF_I)
    ⌘ final inequalities count at reference only
...

```

Similarly we obtain the following by replacing the sort in the algorithm “match on final inequality $<$ ”.

Algorithm “match on final inequality $>$ ”

```

...
V ← Ψ MAT , (NO_MATCH ≤ ROW) , [1.5] (-ROW) ⌘ sorting vector
...
BEG_I ← 1 , v / (1 ↓[1] V) ≠ (¬1 ↓[1] (V ← ¬1 ↓[2] MAT))
    ⌘ inequality in leading columns?
I ← (1 ↓ V) ≠ (¬1 ↓ (V ← , ¬1 ↑[2] MAT))
    ⌘ inequality in final column?
REF_I ← NO_MATCH > ROW
    ⌘ mark references
BEG_I ← BEG_I ∨ (1 , (I ~ 1 ↓ REF_I)
    ∨ ((1 ↓ REF_I) ~ (¬1 ↓ ~ REF_I)))
    ⌘ final inequalities count at reference only,
    final equalities if after data and at reference
...

```

We have created four very similar algorithms to find matches on four inequalities. The whole burden of the solutions rests on the strong shoulders of the brave Grade Up and Grade Down sort primitives Δ and Ψ !

3 Matching with “initial” inequality

This section modifies the problem of section 2 and discusses different kinds of matching in a simple setting. The sorting algorithm “match on equality” of subsection 1.4 can mostly still be used as the basis of a solution. In one case however the limits of the normal sort are reached.

3.1 Cases with initial inequality

In some cases the first property of the desired match cannot be expected to be an exact one. The most common example in German health insurance is the

business time of a “premium capping scheme”. It is not important what the latter exactly is. The point of interest is, that on certain dates $r_i^{(1)}$ a capping scheme is devised as a global property for all plans.

Each capping scheme contains instructions for some, but not all, plans. To answer the question “are the premiums of person j to be capped at time $d_j^{(1)}$ ” the (last) valid capping scheme has to be found and analysed. Should it contain instructions for the plans of the person they are to be used. Otherwise no capping is done.

Mathematically speaking our reference list is a set \mathcal{R} of (distinct) pairs $r = (r^{(1)}, r^{(2)})$ of business times and plans. For a person j with plan $d_j^{(2)}$ at business time $d_j^{(1)}$ we are interested in the subset $\mathcal{R}(d_j)$ defined as

$$r_{max}^{(1)}(d_j) = \max\{r^{(1)}; r = (r^{(1)}, r^{(2)}) \in \mathcal{R} \text{ and } r^{(1)} \leq d_j^{(1)}\},$$

$$\mathcal{R}(d_j) = \{r; r^{(1)} = r_{max}^{(1)}(d_j) \text{ and } r^{(2)} = d_j^{(2)}\} \subset \mathcal{R}.$$

In this section we will consider problems with one first (and therefore “initial”) match on inequality, more specific on the condition “less or equal”, followed by one match on some relation.

On the APL side the first column of our matrices is to be matched on “less or equal” (\leq) whereas the second (and last) one is to be matched on some relation ($=$ or \leq).

3.2 The first type of match on initial inequality

We start with a match of the type we just described. We want to use a sorting algorithm. There does not seem to exist a single sort that would allow us to subsequently group the rows as necessary.

To see this consider REF \leftarrow 2 2 ρ 1 1 2 0 and DAT \leftarrow 3 2 ρ 3 1 4 0 5 1. We expect *no* match for the first and third row of the data, the result should be IND \leftarrow 3 2 3. Neither a sort after the first column, the second one or a combination of both delivers the right groups.

Without a formal proof of correctness we give a solution based on the algorithm “match on final inequality \leq ” but equipped with a loop through the columns. It is relatively clear that it produces the desired result.

Algorithm “example of match on initial inequality”

```

IND  $\leftarrow$  (1  $\uparrow$   $\rho$ DAT)  $\rho$  (NO_MATCH  $\leftarrow$  1 + 1  $\uparrow$   $\rho$ REF)
      A result is 4 byte integer

MAT  $\leftarrow$  REF , [1] DAT Acatenate
ROW  $\leftarrow$   $\iota$  1  $\uparrow$   $\rho$ MAT A member numbers

V  $\leftarrow$   $\mathbb{A}$  MAT , ROW A first sorting vector
MAT  $\leftarrow$  MAT[V;] A sort matrix
ROW  $\leftarrow$  ROW[V ] A sort member numbers

I  $\leftarrow$  (1  $\downarrow$  V)  $\neq$  ( $\bar{1}$   $\downarrow$  (V  $\leftarrow$  MAT[;1])) A inequality in first column?
REF_I  $\leftarrow$  NO_MATCH > ROW A mark references
    
```



```

BEG_I ← 1 , I ← 1 ↓ REF_I
      A inequalities count at reference only

V ← Δ (+ \ BEG_I) , MAT[:,2] , ROW A second sorting vector
MAT ← MAT[V;]                      A sort matrix
ROW ← ROW[V ]                       A sort member numbers
                                      A BEG_I needs no sorting

I ← (1 ↓ V) ≠ (¬1 ↓ (V ← MAT[:,2]))
      A inequality in second column?
BEG_I ← BEG_I ∨ 1 , I
      A all inequalities count

DAT_I ← NO_MATCH ≤ ROW A mark data

V ← BEG_I / ι 1 ↑ ρMAT                A begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ρMAT)) - V A members per group

IND[(1 - NO_MATCH) + DAT_I / ROW]
  ← DAT_I / MAT_NR / NO_MATCH | BEG_I / ROW
      A member number of first match in reference

```

3.3 A different type of match on initial inequality

At the beginning of the section we have described some actuarial data (capping schemes) and its interpretation. However a second interpretation is possible. For a person j with plan $d_j^{(2)}$ at business time $d_j^{(1)}$ we could instead first build the subset

$$\mathcal{R}_0 = \{r; r^{(1)} \leq d_j^{(1)} \text{ and } r^{(2)} = d_j^{(2)}\} \subset \mathcal{R} \quad (3)$$

and based on that select

$$r_{max}^{(1)}(d_j) = \max\{r^{(1)}; r = (r^{(1)}, r^{(2)}) \in \mathcal{R}_0 \text{ and } r^{(1)} \leq d_j^{(1)}\},$$

$$\mathcal{R}(d_j) = \{r; r^{(1)} = r_{max}^{(1)}(d_j) \text{ and } r^{(2)} = d_j^{(2)}\} \subset \mathcal{R}_0.$$

The resulting matches are *not* the ones obtained before. In the example $\mathcal{R} = \{(1,1), (2,0)\}$ and $D = \{(3,1), (4,0), (5,1)\}$ there is now a match for *all* data points, namely (1,1) for the first and third. The procedure does not expect or respect “holes” in data for subsequent business dates.

There is one important technical reason to use this interpretation of “match”. For *one* data point it allows a very effective implementation as (static) SQL³, something like

```

SELECT BUS_TIME, PLAN
FROM CAPPING_SCHEMES
WHERE BUS_TIME <= #B_T AND PLAN = #P

```

³More precisely the statement delivers an unambiguous result if there is only one match in the reference. Normally the data model will guarantee this. If not, the assumption that ORDER BY is stable must be added to achieve uniqueness.

```
ORDER BY BUS_TIME DESC, PLAN ASC
FETCH FIRST 1 ROWS ONLY
```

In this (very) special case we can switch the columns and use the algorithm “match on final inequality \leq ” on the APL side to obtain the same result. This simple example also shows that data modelling must take such questions as the “right” order of the columns and the desired kind of match into account — but that should be self-evident.

3.4 A practically not very important match type

We introduce another match type. It has no obvious practical applications for actuarial simulations in German health insurance, but it puts the two types we already encountered in perspective. It imposes a stronger condition on matches — and will surely have some application!

We consider a very common setting. There are some properties (like insurance number, plan et cetera) that define each policy of a person. Each of them “lives” in system as well as in business time (in the language of DB2 bi-temporal tables).

We ignore the first properties and try to match a (single) bi-temporal data point $d = (d^{(1)}, d^{(2)})$, $\mathcal{D} = \{d\}$, with a (single, whole) policy as reference list \mathcal{R} . As the subset of matches we define

$$r_{max}^{(k)}(d) = \max\{r^{(k)}; r = (r^{(1)}, r^{(2)}) \in \mathcal{R} \text{ and } r^{(k)} \leq d^{(k)}\}, \quad k = 1, 2,$$

$$\mathcal{R}(d) = \{r; r^{(1)} = r_{max}^{(1)}(d) \text{ and } r^{(2)} = r_{max}^{(2)}(d)\} \subset \mathcal{R}.$$

This may not seem “right” (and is probably not the common interpretation of system and business time!), but mathematically it is as sound as the other types of match on inequality we discussed. . .

On the APL side we can find (each) $r_{max}^{(k)}(d)$ through a search for a match on one column. Because the additional requirement to find the first match defines a unique r for each d we can then search and match $(r_{max}^{(k)}(d))_k$ on equality. We can use an algorithm like the following.

Algorithm “example of match on two \leq -inequalities”

```
IND ← (1 ↑ ρDAT) ρ (NO_MATCH ← 1 + 1 ↑ ρREF)
    ⍝ result is 4 byte integer

MAT ← REF ,[1] DAT ⍝ concatenate
ROW ← ι 1 ↑ ρMAT ⍝ member numbers

MATCH_I ← (1 ↑ ρDAT) ρ 1 ⍝ all data points may have matches

:FOR COL :IN ι 1 ↓ ρMAT
    MATP ← MAT[;COL] ⍝ relevant column

    V ← ⍋ MATP ,[1.5] ROW ⍝ sorting vector
    MATP ← MATP[V] ⍝ sort column
```

```

ROWP ← ROW [V]           A sort member numbers

I ← (1 ↓ MATP) ≠ (¯1 ↓ MATP) A inequality?
REF_I ← NO_MATCH > ROWP   A mark references
BEG_I ← 1 , I ~ 1 ↓ REF_I
      A inequalities count at reference only

DAT_I ← NO_MATCH ≤ ROWP A mark data

V ← BEG_I / ⍳ ρMATP           A begin of each group
MAT_NR ← (1 ↓ V , (1 + ρMATP)) - V A members per group

MAT[DAT_I / ROWP ; COL] ← DAT_I / MAT_NR / BEG_I / MATP
      A replace data with extremal values
V ← (1 - NO_MATCH) + DAT_I / ROWP
      A member numbers of data points
I ← NO_MATCH > DAT_I / MAT_NR / NO_MATCH [ BEG_I / ROWP
      A extremal value found?
MATCH_I[V] ← MATCH_I[V] ~ I
      A may still find match?
:ENDFOR

V ← ⍋ MAT , ROW A final sorting vector
MAT ← MAT[V;]   A sort matrix
ROW ← ROW[V ]   A sort member numbers

BEG_I ← 1 , v / (1 ↓ [1] MAT) ≠ (¯1 ↓ [1] MAT)
      A all inequalities count

DAT_I ← NO_MATCH ≤ ROW A mark data

V ← BEG_I / ⍳ 1 ↑ ρMAT           A begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ρMAT)) - V A members per group

IND[(1 - NO_MATCH) + MATCH_I / DAT_I / ROW]
  ← MATCH_I / DAT_I / MAT_NR / NO_MATCH [ BEG_I / ROW
      A member number of first match in reference

```

3.5 A match problematic on the APL side

Now we consider the bi-temporal setting of subsection 3.4 but with a match of the type discussed in subsection 3.3. This is probably a common way to handle such tables.⁴

We consider a trivial reference list with two elements, $\mathcal{R} = \{(1,1), (2,3)\}$ and the single data point $\mathcal{D} = \{(3,2)\}$. The desired match is the first element of the reference list.

If we catenate the two sets APL side the result is (lexicographically) sorted. There is no obvious sort that would make our grouping algorithm work. An APL

⁴Except of course if they are bi-temporal DB2 tables in the specific technical meaning of IBM, in which case they are completely system handled!

algorithm could of course first try to find out which rows fulfil the necessary inequalities (those that define the analog of subset \mathcal{R}_0 in definition (3)).

The answer however depends on the data point. It is not obvious how an effective and fast algorithm could be build around this idea in APL.

3.6 Some remarks on matches and their different types

The standard matches (only on equality) are much easier than the ones containing inequalities. The predicative conditions on the columns may be examined one after another in an arbitrary order (commutativity). They may also be examined simultaneously.

When inequalities are required not only is the sequence in which they are examined important but also the type of the match used. We have encountered three match types. In each case there must be an explicit agreement on what data model is used — the data itself does not provide the necessary information.

On the APL side we have a similar situation. Each inequality brings more problems and makes the necessary algorithms (much) more complicated and slow. In subsection 3.5 we encountered a setting with no obvious APL solution.

Finding matches can be a goal itself. It can however also form the kernel of many other algorithms. In particular matches on inequality may be very useful even if the reference has keys defined in bounded intervals. It can be very useful to match the lower bound on inequality \leq and then compare the data point with the upper bound — always assuming of course that the reference is overlap free!

4 Matching on set equality and relation

This section generalizes and formalizes the matching problem. It defines and discusses different kinds of matching on set equality and relation. It is shown that each matching problem corresponds to a disjoint dissection of the direct sum of reference and data list.

4.1 Setting

Let \mathcal{S}_0 be a set (taken to mean “with distinct elements”) equipped with a relation $<$ under which it is completely ordered. One may think of this base set \mathcal{S}_0 being \mathbb{Z} or \mathbb{R} . In fact we are mainly interested in their special subsets that are presentable on a digital computer.

We are interested in finite groups \mathcal{S} of elements of \mathcal{S}_0^n , $n \geq 1$. We equip \mathcal{S} with the set equality $=$ (in particular it may contain identical elements) and use $<$, \leq , $>$ and \geq in the obvious way for each coordinate $s^{(k)}$ of each $s \in \mathcal{S}$.

In particular the quotient \mathcal{S}/\equiv is a set. We (try to) use “group”, “subgroup” and “(group) member” when we refer to \mathcal{S} and its members (!). In contrast we use “set”, “subset” and “(set) element” when we refer to equivalence classes of members or elements of \mathcal{S}_0^n .

We fix a “dimension” n , select two corresponding groups and call the one “reference”, denoted by \mathcal{R} , and the other “data”, denoted by \mathcal{D} . We number the members of $\mathcal{R} = \{r_1, \dots, r_i, \dots, r_l\}$, $l = |\mathcal{R}|$, in an arbitrary but fixed way. This is done in order to achieve unique matches (to give the phrase “the *first* match” meaning).

4.2 Matches in general

We fix a vector $\leftrightarrow = (\leftrightarrow^{(k)})_k \in \{=, <, \leq, >, \geq\}^n$, of relations on \mathcal{R} , \mathcal{D} and $\mathcal{M} = \mathcal{R} \oplus \mathcal{D}$. We use

$$m_i \leftrightarrow m_{i'} \quad :\Leftrightarrow \quad m_i^{(k)} \leftrightarrow^{(k)} m_{i'}^{(k)} \text{ for all } k = 1, \dots, n,$$

for members of \mathcal{M} .

Associating pro forma the relation $=$ with the order of $<$ (for this relation “sort” is taken to mean “usual ascending sort”) and extending naturally we define a corresponding lexicographic relation and order on $(\mathcal{R}, \mathcal{D})$ and \mathcal{M} . We use

$$m_i \leftrightarrow_{lex} m_{i'} \quad :\Leftrightarrow \quad m_i^{(k)} \leftrightarrow^{(k)} m_{i'}^{(k)} \text{ for } k = \min\{k'; m_i^{(k')} \neq m_{i'}^{(k')}\},$$

for elements of \mathcal{M} .

Of course the defined relations cannot distinguish identical members

$$m_i = m_{i'} \quad \Rightarrow \quad m_i \leftrightarrow m_{i'}, m_i \leftrightarrow_{lex} m_{i'}, m_{i'} \leftrightarrow m_i \text{ and } m_{i'} \leftrightarrow_{lex} m_i$$

For each member $d \in \mathcal{D}$ we seek “the first match” $r \in \mathcal{R}$. More specifically we consider three types of match. In all cases we define filtered groups of members with the last one, $\mathcal{R}^T(d)$, containing exactly the matches for d . Those of course obey all relations with respect to d

$$\begin{aligned} \mathcal{R} \supseteq \mathcal{R}_0^T(d) \supseteq \mathcal{R}_1^T(d) \supseteq \dots \supseteq \mathcal{R}_n^T(d) = \mathcal{R}^T(d) \\ \mathcal{R}^T(d) \subseteq \mathcal{R}_{\leftrightarrow}(d) = \{r \in \mathcal{R}; r \leftrightarrow d\} \end{aligned}$$

More specifically $\mathcal{R}^T(d)$ contains at most one element of \mathcal{S}_0^n (all contained members are identical). In all cases we select $r_i \in \mathcal{R}^T(d)$ with $i = \min\{i'; r_{i'} \in \mathcal{R}^T(d)\}$ as the desired (first) match and denote it by its number i . In case $\mathcal{R}^T(d) = \emptyset$ is empty we define $i = 1 + |\mathcal{R}|$.

4.3 Definition of some interesting types of match

The types of match that interest us are the following. With $\text{extr}\{\ast\}$ we mean the obvious “extremal” value ($=, \min, \max$) under the relevant relation. A match is “weak” when it is sought only under references fulfilling a starting condition, “strong” otherwise.

In some cases the extremal values are “global” and may that fore be defined simultaneously whereas in others they are “local” and may not. Note that in all cases the filtered subgroups are defined successively, although in the case of global extrema the final subgroup can be defined directly. The selection of “the first match” is a possible add on in all cases.

weak local We start with $\mathcal{R}_0^{wk.loc}(d) = \mathcal{R}_{\leftrightarrow}(d)$ and define successively for $k = 1, \dots, n$

$$\begin{aligned} r_{extr}^{(k)}(d) &= \text{extr}\{r^{(k)}; r \in \mathcal{R}_{k-1}^{wk.loc} \text{ and } r^{(k)} \leftrightarrow^{(k)} d^{(k)}\}, \\ \mathcal{R}_k^{wk.loc}(d) &= \{r \in \mathcal{R}_{k-1}^{wk.loc}; r^{(k)} = r_{extr}^{(k)}(d)\} \subset \mathcal{R}_{k-1}^{wk.loc}. \end{aligned}$$

$1, \dots, n$

$$r_{extr}^{(k)}(d) = \text{extr}\{r^{(k)}; r \in \mathcal{R}_{k-1}^{str,loc} \text{ and } r^{(k)} \leftrightarrow^{(k)} d^{(k)}\},$$

$$\mathcal{R}_k^{str,loc}(d) = \{r \in \mathcal{R}_{k-1}^{str,loc}; r^{(k)} = r_{extr}^{(k)}(d)\} \subset \mathcal{R}_{k-1}^{str,loc}.$$

weak global We start with $\mathcal{R}_0^{wk,gl}(d) = \mathcal{R}_{\leftrightarrow}(d)$ and define successively for $k = 1, \dots, n$

$$r_{extr}^{(k)}(d) = \text{extr}\{r^{(k)}; r \in \mathcal{R}_{\leftrightarrow}(d) \text{ and } r^{(k)} \leftrightarrow^{(k)} d^{(k)}\} = \text{extr}\{r^{(k)}; r \in \mathcal{R}_{\leftrightarrow}(d)\},$$

$$\mathcal{R}_k^{wk,gl}(d) = \{r \in \mathcal{R}_{k-1}^{wk,gl}; r^{(k)} = r_{extr}^{(k)}(d)\} \subset \mathcal{R}_{k-1}^{wk,gl}.$$

strong global We start with $\mathcal{R}_0^{str,gl}(d) = \mathcal{R}$ and define successively for $k = 1, \dots, n$

$$r_{extr}^{(k)}(d) = \text{extr}\{r^{(k)}; r \in \mathcal{R} \text{ and } r^{(k)} \leftrightarrow^{(k)} d^{(k)}\},$$

$$\mathcal{R}_k^{str,gl}(d) = \{r \in \mathcal{R}_{k-1}^{str,gl}; r^{(k)} = r_{extr}^{(k)}(d)\} \subset \mathcal{R}_{k-1}^{str,gl}.$$

The weak local match is also “lexicographic” because, apart from the obvious condition imposed by the relation \leftrightarrow , it involves only a lexicographic sort. For each (but only *one!*) d the match⁵ may be found with a SQL statement of the form

```
SELECT FIELD1, FIELD2, ..., FIELDN
FROM TABLE
WHERE FIELD1 <= #D1
AND   FIELD2  = #D2
...
AND   FIELDn >= #Dn
ORDER BY FIELD1 DESC, FIELD2 DESC, ..., FIELDn ASC
FETCH FIRST 1 ROWS ONLY
```

As a filler we have used $\leftrightarrow = (\leq, =, \dots, \geq)$ in the statement. We could of course have used “FIELD2 ASC” in the ORDER BY clause as well.

The strong local match is better suited to a non-technical setting. It assumes that there are some properties ordered after descending importance. Each one defines the subgroup (in a meaningful real world setting probably a subset...) of values relevant for the rest.

If all properties are “hole-less”, meaning that each key used in the k -th position of one combination of keys appears at the k -th position of some combination of keys for *all* combinations of keys without their k -th position, then the difference with weak local matches disappears.

⁵More precisely the statement delivers a member of the subgroup of matches. Normally the data model will guarantee that there is exactly one such member. If not, the assumption that ORDER BY is stable must be added if we want to get the *same* member each time.

4.4 Analysis of the different types of match

4.4.1 Comparison and true inclusions

As already mentioned, groups of matches are either one element sets or empty because extremal values are unique. Let now $d \in \mathcal{D}$ be arbitrary but fixed.

If there exists $r \in \mathcal{R}^{str,gl}(d)$ all its coordinates are global extremal values. That fore they are also extremal in every subgroup. Furthermore $r \in \mathcal{R}_{\leftrightarrow}(d)$ holds. It follows that $r \in \mathcal{R}_0^{wk,loc}(d) \cap \mathcal{R}_0^{str,loc}(d)$ also holds.

Inductively $r \in \mathcal{R}_k^{wk,loc}(d) \cap \mathcal{R}_k^{str,loc}(d)$ holds for each k . It follows that r is also an member of $\mathcal{R}^{wk,loc}(d)$ and $\mathcal{R}^{str,loc}(d)$. That fore $\mathcal{R}^{str,gl}(d) \subset \mathcal{R}^{wk,loc}(d) \cap \mathcal{R}^{str,loc}(d)$ holds.

Now let $r \in \mathcal{R}^{str,loc}(d)$ exist. In particular $r \in \mathcal{R}_{\leftrightarrow}(d)$ holds. Because $r^{(1)}$ is globally extremal it is also with respect to $\mathcal{R}_0^{wk,loc}(d) = \mathcal{R}_{\leftrightarrow}(d)$ and so $r \in \mathcal{R}_1^{wk,loc}(d)$ also holds.

Inductively the same argument can be used and $r \in \mathcal{R}_k^{wk,loc}(d)$ holds for each k . It follows that r is also an member of $\mathcal{R}^{wk,loc}(d)$. That fore $\mathcal{R}^{str,loc}(d) \subset \mathcal{R}^{wk,loc}(d)$ holds.

Now let $r \in \mathcal{R}^{str,gl}(d)$ (again) exist. In particular $r \in \mathcal{R}_{\leftrightarrow}(d)$ holds. Because each $r^{(k)}$ is globally extremal it is also with respect to $\mathcal{R}_0^{wk,gl}(d) = \mathcal{R}_{\leftrightarrow}(d)$ and so $r \in \mathcal{R}^{wk,gl}(d)$ also holds.

Finally let $r \in \mathcal{R}^{wk,gl}(d)$ exist. As each $r^{(k)}$ is globally extremal with respect to $\mathcal{R}_0^{wk,loc}(d) = \mathcal{R}_{\leftrightarrow}(d)$ it is also with respect to $\mathcal{R}_k^{wk,loc}(d)$ and so $r \in \mathcal{R}^{wk,loc}(d)$ also holds.

For each $d \in \mathcal{D}$ have established

$$\begin{aligned} \mathcal{R}^{wk,loc}(d) &\supseteq \mathcal{R}^{str,loc}(d) \supseteq \mathcal{R}^{str,gl}(d) \\ \mathcal{R}^{wk,loc}(d) &\supseteq \mathcal{R}^{wk,gl}(d) \supseteq \mathcal{R}^{str,gl}(d). \end{aligned}$$

Now let T and T' be two different types of match, $d \in \mathcal{D}$ arbitrary but fix with a match of *both* types, $\mathcal{R}^T(d) \neq \emptyset$ and $\mathcal{R}^{T'}(d) \neq \emptyset$. We recall the fact that $\mathcal{R}^T(d) \cap \mathcal{R}^{T'}(d) \subseteq \mathcal{R}_{\leftrightarrow}(d)$ and use it repeatedly.

If $\mathcal{R}_0^T(d) = \mathcal{R}_0^{T'}(d)$ then of course the extremal values $r_{extr}^{T,(1)}(d) = r_{extr}^{T',(1)}(d)$ are equal. Else let $r_{extr}^{T,(1)}(d) \leftrightarrow r_{extr}^{T',(1)}(d)$ hold. But then there exists $r \in \mathcal{R}_{\leftrightarrow}(d)$ with $r^{(1)} = r_{extr}^{T,(1)}(d)$ (by definition of $\mathcal{R}^T(d)$) and $r^{(1)} \leftrightarrow r_{extr}^{T',(1)}(d)$ (by definition of $\mathcal{R}_{\leftrightarrow}(d)$), so $r_{extr}^{T,(1)}(d) \leftrightarrow r_{extr}^{T',(1)}(d)$ and that fore $r_{extr}^{T,(1)}(d) = r_{extr}^{T',(1)}(d)$ follows.

Now let $r_{extr}^{T,(k')} (d) = r_{extr}^{T',(k')} (d)$ hold for all $k' = 1, \dots, k-1$. Similarly to the case $k=1$ we can find an $r \in \mathcal{R}_{\leftrightarrow}(d)$ with

$$\begin{aligned} r^{(k')} &= r_{extr}^{T,(k')} (d) = r_{extr}^{T',(k')} (d), \quad \text{for all } k' < k \\ r^{(k)} &= r_{extr}^{T',(k)} (d) \\ r^{(k)} &\leftrightarrow r_{extr}^{T,(k)} (d) \end{aligned}$$

so $r_{extr}^{T',(k)}(d) \leftrightarrow r_{extr}^{T,(k)}(d)$ and then $r_{extr}^{T,(k)}(d) = r_{extr}^{T',(k)}(d)$ follow.

For each $d \in \mathcal{D}$ and each pair T and T' of match types we have established

$$\mathcal{R}^T(d) \neq \emptyset \quad \text{and} \quad \mathcal{R}^{T'}(d) \neq \emptyset \quad \Rightarrow \quad \mathcal{R}^T(d) = \mathcal{R}^{T'}(d).$$

4.4.2 False inclusions

Evidently *neither* the inverse *nor* additional inclusions are true. Some absolutely trivial examples are the following:

- $\mathcal{R} = \{(1, 1), (2, 3)\}$, $d = (3, 2)$ and $\leftrightarrow = (\leq, \leq)$. Here $\mathcal{R}^{wk,loc}(d) = \{(1, 1)\}$ whereas $\mathcal{R}^{str,loc}(d) = \emptyset$.
- $\mathcal{R} = \{(3, 0), (0, 3)\}$, $d = (4, 4)$ and $\leftrightarrow = (\leq, \leq)$. Here $\mathcal{R}^{str,loc}(d) = \{(3, 0)\}$ whereas $\mathcal{R}^{str,gl}(d) = \emptyset$.
- $\mathcal{R} = \{(1, 2), (2, 1)\}$, $d = (3, 2)$ and $\leftrightarrow = (\leq, \leq)$. Here $\mathcal{R}^{wk,loc}(d) = \{(2, 1)\}$ whereas $\mathcal{R}^{wk,gl}(d) = \emptyset$.
- $\mathcal{R} = \{(3, 0), (5, 3)\}$, $d = (4, 4)$ and $\leftrightarrow = (\leq, \leq)$. Here $\mathcal{R}^{wk,gl}(d) = \{(3, 0)\}$ whereas $\mathcal{R}^{str,gl}(d) = \emptyset$.
- $\mathcal{R} = \{(1, 1), (2, 3)\}$, $d = (3, 2)$ and $\leftrightarrow = (\leq, \leq)$. Here $\mathcal{R}^{wk,gl}(d) = \{(1, 1)\}$ whereas $\mathcal{R}^{str,loc}(d) = \emptyset$.
- $\mathcal{R} = \{(1, 2), (2, 1)\}$, $d = (3, 2)$ and $\leftrightarrow = (\leq, \leq)$. Here $\mathcal{R}^{str,loc}(d) = \{(2, 1)\}$ whereas $\mathcal{R}^{wk,gl}(d) = \emptyset$.

4.4.3 “Commutativity” of relations

We now analyse what happens if we work through $k \in \{1, \dots, n\}$ in an order different than the natural one. Do the resulting matches $\mathcal{R}^T(d)$ stay the same? In some cases the order of the coordinates is important and in others not.

weak local The definition of $\mathcal{R}_0^{wk,loc}(d) = \mathcal{R}_{\leftrightarrow}(d)$ does not use the order of the coordinates. Moreover (if not empty) all its members have the same $r^{(k)}$ for each k with $\leftrightarrow^{(k)}$ an equality. Therefore the corresponding extremal values are fixed and equalities may be processed in an arbitrary order *before* all inequalities. This freedom includes the process of building $\mathcal{R}_0^{wk,loc}(d)$ itself.

On the other hand the trivial example $\mathcal{R} = \{(3, 2), (4, 4)\}$, $d = (3, 4)$ and $\leftrightarrow = (=, \leq)$ shows that it is not possible to completely process the inequalities (including during the building of $\mathcal{R}_0^{wk,loc}(d)$) *first*, because this would give $\mathcal{R}^{wk,loc}(d) = \emptyset$, whereas $\mathcal{R}^{wk,loc}(d) = \{(3, 2)\}$ is the correct answer. Of course once $\mathcal{R}_0^{wk,loc}(d)$ has been build equalities may also be processed after inequalities.

Furthermore the example $\mathcal{R} = \{(0, 3), (3, 0)\}$, $d = (4, 4)$ and $\leftrightarrow = (\leq, \leq)$ shows that it is not possible to process inequalities in an arbitrary order, because a switch would give $\mathcal{R}^{wk,loc}(d) = \{(0, 3)\}$, whereas $\mathcal{R}^{wk,loc}(d) = \{(3, 0)\}$ is the correct answer.

strong local The examples $\mathcal{R} = \{(3, 2), (4, 4)\}$, $d = (3, 4)$ and $\leftrightarrow = (=, \leq)$ as well as $\mathcal{R} = \{(0, 3), (3, 0)\}$, $d = (4, 4)$ and $\leftrightarrow = (\leq, \leq)$ we just used for weak local matches also show that it is neither possible to switch equalities and inequalities nor to process inequalities in an arbitrary order when searching strong local matches.

weak global The definition of $\mathcal{R}_0^{wk,gl}(d) = \mathcal{R}_{\leftrightarrow}(d)$ does not use the order of the coordinates. Furthermore the extremal values are defined as global ones on $\mathcal{R}_0^{wk,gl}(d)$. They are thus of course independent of each other. Therefore $\mathcal{R}^{wk,gl}(d)$ is completely independent of the order in which the coordinates are processed.

strong global The extremal values are defined as global ones. They are thus of course independent of each other. Therefore $\mathcal{R}^{str,gl}(d)$ is completely independent of the order in which the coordinates are processed.

4.5 Disjoint dissections

Now we consider two members $d, d' \in \mathcal{D}$ and their associated filtered groups $\mathcal{R}_k^T(d)$ and $\mathcal{R}_k^T(d')$. For $k = 1$, and then inductively for all k , d and d' are associated either with the same extremal value or not. As a result either $\mathcal{R}_k^T(d) = \mathcal{R}_k^T(d')$ or $\mathcal{R}_k^T(d) \cap \mathcal{R}_k^T(d') = \emptyset$ holds.

If $d = d'$ then of course the first is the case. We get disjoint dissections of $\mathcal{D} = \cup \mathcal{D}_i^T$ and $\mathcal{R} = \cup \mathcal{R}_i^T$ of data points and associated matches. Of course there are subgroups \mathcal{D}_i^T associated with \emptyset as well as subgroups \mathcal{R}_i^T without “interested” data points. The dissections are also disjoint dissections of the underlying sets.

Let us apply the result to the direct sum $\mathcal{M} = \mathcal{D} \oplus \mathcal{R}$. We have created a disjoint dissection of $\mathcal{M} = \cup \mathcal{M}_i^T$ such that the matches of each $d \in \mathcal{M}_i^T \cap \mathcal{D}$ are to be found in $\mathcal{M}_i^T \cap \mathcal{R}$ (as long as there are any).

The created dissection is (as already remarked in subsection 3.6) useful even if the reference values are not given as begin or end points but rather as intervals. The dissection finds (of course assuming that the correct data model is used. . .) the (only) potentially right interval using only one end and avoiding Cartesian products. The other end of the interval can then be easily used to check if the match is the desired (or else there is no match).

5 Algorithms for two kinds of match

In this section sorting algorithms to find strong (local and global) matches are given. Additionally possible implementations in APL are presented.

5.1 A sorting algorithm for strong local matches

5.1.1 Description

We now reproduce the dissection of the direct sum \mathcal{M} for strong local matches introduced in section 4 (in particular subsection 4.3) via a sorting algorithm than can be implemented in APL. As a “base dissection” we use $\mathcal{M} = \mathcal{M}_1^{(0)}$ and create filtered dissections $\mathcal{M} = \cup \mathcal{M}_i^{(k)}$. In a last step we find the (index of the) first match (for each $d \in \mathcal{D}$).

In the APL algorithm we write down we use both ascending and descending sorting (through APL primitives). We try however to avoid a rather high runtime penalty for highly non-trivial sorting and much data shuffling. To achieve this some additional information must be processed.

It is of course possible (and faster) to use only ascending sorting. The presented code is however easier to read. It is also much easier to verify its correctness (the fact that it delivers the desired strong local matches).

5.1.2 Algorithm

The variables REF and DAT are matrices and represent of course \mathcal{R} and \mathcal{D} respectively, REL is an alphanumeric vector and stands for the relation \leftrightarrow . We assume $\mathcal{S}_0 = \mathbb{Z}$, so that \uparrow respects the natural relation $<$. For $\mathcal{S}_0 = \mathbb{R}$ the comparison tolerance must possibly be set to 0, alphanumeric data can be processed similarly.

Algorithm “strong local match”

```

IND ← (1 ↑ ρDAT) ρ (NO_MATCH ← 1 + 1 ↑ ρREF)
    A result is 4 byte integer

MAT ← REF , [1] DAT A concatenate
ROW ← ι 1 ↑ ρMAT A member numbers

BEG_I ← 1 , ((~1 + 1 ↑ ρMAT) ρ 0) A start with one group

:FOR COL :IN ι 1 ↓ ρMAT
    GR_NR ← + \ BEG_I A group number
    REF_I ← NO_MATCH > ROW A mark references
    :SELECT REL[COL] A build next sorting vector
    :CASE '='
        V ← ↑ GR_NR , MAT[;COL] , (~REF_I) , [1.5] ROW
    :CASE '≤'
        V ← ↑ GR_NR , MAT[;COL] , (~REF_I) , [1.5] ROW
    :CASE '<'
        V ← ↑ GR_NR , MAT[;COL] , REF_I , [1.5] ROW
    :CASE '≥'
        V ← ↓ (-GR_NR) , MAT[;COL] , REF_I , [1.5] (-ROW)
    :CASE '>'
        V ← ↓ (-GR_NR) , MAT[;COL] , (~REF_I) , [1.5] (-ROW)
    :ENDSELECT
    MAT ← MAT[V;] A sort matrix
    ROW ← ROW[V ] A sort member numbers
        A BEG_I needs no sorting

I ← (1 ↓ V) ≠ (~1 ↓ (V ← MAT[;COL])) A inequality?
:SELECT REL[COL]
    A further dissect each subgroup
:CASE '='
    BEG_I ← BEG_I ∨ (1 , I)
    A all inequalities count
:CASELIST '≤≥'
    REF_I ← NO_MATCH > ROW
    BEG_I ← BEG_I ∨ (1 , I ~ 1 ↓ REF_I)
    
```

```

      A inequalities count at reference only
:CASELIST '<>'
  REF_I ← NO_MATCH > ROW
  BEG_I ← BEG_I ∨ (1 , (I ~ 1 ↓ REF_I)
    ∨ ((1 ↓ REF_I) ~ (¬1 ↓ ~ REF_I)))
      A inequalities count at reference only,
      equalities if after data and at reference
:ENDSELECT
:ENDFOR

V ← ⍋ (+ \ BEG_I) , [1.5] ROW
  A ensure sort up and leading reference members
MAT ← MAT[V;]          A sort matrix
ROW ← ROW[V ]         A sort member numbers

DAT_I ← NO_MATCH ≤ ROW A mark data

V ← BEG_I / ⍋ 1 ↑ ρMAT          A begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ρMAT)) - V A members per group

IND[(1 - NO_MATCH) + DAT_I / ROW]
  ← DAT_I / MAT_NR / NO_MATCH [ BEG_I / ROW
    A member number of first match in reference

```

5.1.3 Proof of correctness

In the main part of the algorithm the subgroups are created as defined in subsection 4.3. To achieve this the data is processed column-wise. It is sorted using:

GR_NR This part ensures that already defined subgroups are not teared apart and get not moved around. This also means, that **BEG_I** needs not to be sorted or reset, even if members in the subgroup get reshuffled.

MAT[;COL] This is the data proper. It is clear that it is necessary and sufficient to sort it in order to respect the relevant relations.

REF_I The marking of reference points ensures the desired behaviour whenever the column data proper is equal (meaning in particular that references and data points are indistinguishable in this column). In some of the cases the necessary information is also encoded in **ROW** and the variable could have been omitted.

ROW The use of the original member numbers (coded as rows in APL) guarantees that the sort is stable even if the algorithm underlying the primitive is not. This ensures that the first match lands at the beginning of its subgroup.

Then the already constructed subgroups are further dissected. A case by case analysis makes clear that the algorithm works:

= In this case each inequality defines a new group as it should be. Reference points come before equal data points due to the sorting used.

- ≤ The maximum is unique (for each data point). That fore unequal reference points define new subgroups. Not so for data points, as long as they point to the same maximal reference value. The sorting ensures that reference points come before equal data points.
- < In contrast to ≤ data points must come *before* equal reference points, so they are preceded (only) by strictly smaller references. Additionally the transition between equal data and reference points must define a new subgroup.
- ≥ Analog to ≤.
- > Analog to <.

So it is clear that the correct subgroups are created. The last part selects the first member of each group and uses it to define the first match of each $d \in \mathcal{D}$. To achieve this it ensures that reference points precede equal data points and uses the number of members per subgroup.

5.2 A sorting algorithm for strong global matches

5.2.1 Introduction

We now reproduce the dissection of the direct sum \mathcal{M} for strong global matches introduced in section 4 (in particular subsection 4.3) via a sorting algorithm than can be implemented in APL. As a “base dissection” we use $\mathcal{M} = \mathcal{M}_1^{(0)}$.

We do not follow the definition of the dissection strictly but instead replace data points for inequalities with extremal values coordinate-wise. Based on those new values we seek matches on equality and directly create the final dissection $\mathcal{M} = \cup \mathcal{M}_i^{(n)}$ and find the (index of the) first match (for each $d \in \mathcal{D}$).

For the necessity of this last step consider $\mathcal{R} = \{(3, 0), (0, 3), (3, 3)\}$, $\mathcal{D} = \{(4, 4)\}$ and $\leftrightarrow = (\leq, \leq)$. The search for the maximal value yields matches at the first (as well as the third) and second (as well as the third) member respectively. The desired overall match however is the third member (even more $\mathcal{R}^{str,gl}(d) = (3, 3)$ contains only this one member).

It should be clear that the overall process is equivalent to the definition of strong global matches. We only sort data up (via **4**). This means that when replacing values under the relations \geq or $>$ we must use last members of groups (instead of first ones).

5.2.2 Algorithm

As in subsection 5.1 the variables REF and DAT represent \mathcal{R} and \mathcal{D} respectively and REL stands for the relation \leftrightarrow . The remarks about \mathbb{Z} , \mathbb{R} and alphanumeric data hold.

Algorithm “strong global match”

```
IND ← (1 ↑ ρDAT) ρ (NO_MATCH ← 1 + 1 ↑ ρREF)
  A result is 4 byte integer
```

```
MAT ← REF ,[1] DAT A catenate
```

```

ROW ← ⍒ 1 ↑ ρMAT  ⌘ member numbers

MATCH_I ← (1 ↑ ρDAT) ρ 1 ⌘ all data points may have matches

:FOR COL :IN ('=' ≠ REL) / ⍒ 1 ↓ ρMAT
  ⌘ find extremal values at inequalities
  MATP ← MAT[;COL] ⌘ relevant column

  REF_I ← NO_MATCH > ROW ⌘ mark references
  :SELECT REL[COL]          ⌘ build next sorting vector
  :CASELIST '≤>'
    V ← ⌘ MATP , (~REF_I) , [1.5] ROW
  :CASELIST '<≥'
    V ← ⌘ MATP , REF_I , [1.5] ROW
  :ENDSELECT
  MATP ← MATP[V] ⌘ sort column
  ROWP ← ROW [V] ⌘ sort member numbers

  I ← (1 ↓ MATP) ≠ (~1 ↓ MATP) ⌘ inequality?
  REF_I ← NO_MATCH > ROWP ⌘ mark references
  :SELECT REL[COL] ⌘ dissect on one inequality
  :CASE '≤'
    GR_I ← 1 , (I ~ 1 ↓ REF_I)
    ⌘ inequalities count at reference only
  :CASE '<'
    GR_I ← 1 , ((I ~ 1 ↓ REF_I)
    v ((1 ↓ REF_I) ~ (~1 ↓ ~ REF_I)))
    ⌘ inequalities count at reference only,
    equalities if after data and at reference
  :CASE '≥'
    GR_I ← (I ~ ~1 ↓ REF_I) , 1
    ⌘ inequalities count at reference only
  :CASE '>'
    GR_I ← ((I ~ ~1 ↓ REF_I)
    v ((1 ↓ ~ REF_I) ~ (~1 ↓ REF_I))) , 1
    ⌘ inequalities count at reference only,
    equalities if after reference and at data
  :ENDSELECT

  DAT_I ← NO_MATCH ≤ ROWP ⌘ mark data

  V ← GR_I / ⍒ ρMATP ⌘ begin or end of each group
  :SELECT REL[COL] ⌘ members per group
  :CASELIST '≤<'
    MAT_NR ← (1 ↓ V , (1 + ρMATP)) - V
  :CASELIST '≥>'
    MAT_NR ← V - (0 , ~1 ↓ V)
  :ENDSELECT

```

```

MAT[DAT_I / ROWP ; COL] ← DAT_I / MAT_NR / GR_I / MATP
    A replace data with extremal values
V ← (1 - NO_MATCH) + DAT_I / ROWP
    A member numbers of data points
I ← NO_MATCH > DAT_I / MAT_NR / NO_MATCH [ GR_I / ROWP
    A extremal value found?
MATCH_I[V] ← MATCH_I[V] ^ I
    A may still find match?
:ENDFOR

V ← Δ MAT , ROW A final sorting vector
MAT ← MAT[V;] A sort matrix
ROW ← ROW[V ] A sort member numbers

BEG_I ← 1 , v / (1 ↓[1] MAT) ≠ (-1 ↓[1] MAT)
    A all inequalities count

DAT_I ← NO_MATCH ≤ ROW A mark data

V ← BEG_I / ι 1 ↑ ρMAT A begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ρMAT)) - V A members per group

IND[(1 - NO_MATCH) + MATCH_I / DAT_I / ROW]
    ← MATCH_I / DAT_I / MAT_NR / NO_MATCH [ BEG_I / ROW
    A member number of first match in reference
    
```

5.2.3 Proof of correctness

There is not much to prove. One must only think each case (relation) through and make sure that the sort and the following definition of subgroups are the right ones for finding the wanted extremal value. We refer to subsection 5.1 for details.

The last step is to define the (final) dissection. More precisely the dissection on equalities must be further divided. At this point it is important to be sure that (all) references come either before or after (all) corresponding data points. That fore a mix of sort up and down must be used in the main part. The use of ROW ensures that our matches are first ones.

5.2.4 An alternative approach

There is another possible approach to a solution. It uses the fact, proven in subsection 4.4, that matches, as far as they exists, do not depend on their type. That fore we can try to find $\mathcal{R}^{str,gl}(d)$ starting with $\mathcal{R}^{str,loc}(d)$.

The algorithm "strong local match" presented in subsection 5.1 sorts reference and data in such a way, that the disjoint dissection necessary for strong local matches is created. Let $d \in \mathcal{D}$ be arbitrary but fix. It is contained in a subgroup $\mathcal{M}(d)$.

If and only if there exists a $r \in \mathcal{R} \cap \mathcal{M}(d)$ there is a strong local match ($\mathcal{R}^{str,loc}(d) \neq \emptyset$). Without loss of generality we assume that r is the first

match. Because of $\mathcal{R}^{str,loc}(d) \supseteq \mathcal{R}^{str,gl}(d)$ this is necessary but not sufficient for the existence of a strong global match ($\mathcal{R}^{str,gl}(d) \neq \emptyset$).

We indicate the potential for a match as $\mathbb{1}_0^{str,gl}(d)$. For each coordinate we successively examine if $r^{(k)}$ is also globally extremal. If not then there exists no match. In formulas

$$\mathbb{1}_k^{str,gl}(d) = \begin{cases} 1 & \text{if } k = 0 \text{ and } \mathcal{R}^{loc,gl}(d) \neq \emptyset \\ \mathbb{1}_{k-1}^{str,gl}(d) & \text{if } k \geq 1 \text{ and} \\ & r^{(k)} = \text{extr}\{(r')^{(k)}; r' \in \mathcal{R} \text{ and } (r')^{(k)} \leftrightarrow^{(k)} d^{(k)}\} \\ 0 & \text{else} \end{cases}$$

If and only if $\mathbb{1}_n^{str,gl}(d)$ is true there exists a strong global match ($\mathcal{R}^{str,gl}(d) \neq \emptyset$) and (then) r is the first match. We used a particular order to check the conditions of extremality but it is obviously arbitrary and not relevant for the end result. Furthermore each sort can clearly be applied to an arbitrary ordering of \mathcal{M} .

A possible algorithmic implementation in APL is obviously to use the algorithm "strong local match" with an additional sort for each column k as an add on. The latter is used to determine if $r^{(k)}$ is globally extremal with respect to d . However the presented algorithm "strong global match" is surely less circuitous and more efficient.

6 Tentative algorithms for problematic matches

This section outlines sorting algorithms for weak (local and global) matches. There is however no obvious way to implement them effectively using APL primitives. "Special sorts" are proposed as a solution to the problem.

6.1 A tentative sorting algorithm for weak local matches

6.1.1 Introduction

We now want to reproduce the dissection of \mathcal{M} for weak local matches via a sorting algorithm than can be implemented in APL. As a "base dissection" we use $\mathcal{M} = \mathcal{M}_1^{(0)}$.

We want to follow the definition of the dissection. To achieve this we want to use a "special sort" of the group \mathcal{M} to enforce that all $d \in \mathcal{D}$ which have the necessary relation to a specific $r \in \mathcal{R}$ (including extremality) and only those follow this r (directly). This given it is trivial to directly create the final dissection $\mathcal{M} = \bigcup_i \mathcal{M}_i^{(n)}$.

It should be clear that the overall process is equivalent to the definition of weak local matches. What is unclear is how to sort the data using the APL primitives \blacktriangle and \blacktriangledown .

6.1.2 Algorithm with a gap

As in subsection 5.1 the variables REF and DAT represent \mathcal{R} and \mathcal{D} respectively and REL stands for the relation \leftrightarrow . The remarks about \mathbb{Z} , \mathbb{R} and alphanumeric data hold.

We use the obvious fact (see also subsection 4.4) that equalities may be tested at the beginning of the algorithm. The intermediate dissection thus created is only a technical trick to separate those and has not much in common with the ones in the definition of the match.

We write SPEC_SORT as a place holder for a special type of sort. It uses a boolean flag vector in addition to the data proper. The algorithm sorts data in a way that all but creates the desired final dissection (all members are correctly “positioned” and only the beginning of each group must be determined).

To do this it must ensure that (all) references come either before or after (all) corresponding data points. That fore sort up and down may both be needed. Note that the resulting order of each sort is the basis of the next one but the subgroups corresponding to each sort are not and *cannot be* used subsequently.

Algorithm “weak local match”

```

IND ← (1 ↑ ρDAT) ρ (NO_MATCH ← 1 + 1 ↑ ρREF)
    A result is 4 byte integer

MAT ← REF ,[1] DAT A concatenate
ROW ← ι 1 ↑ ρMAT A member numbers

COL_EQ ← (I ← '=' = REL) / (V ← ι ρREL) A separate equalities
COL_INEQ ← (I ← '~I') / V A and inequalities

V ← (2 / 1 ^-1) ['<=>'] ι I / REL A sign
REF_I ← NO_MATCH > ROW A mark references
V ← Δ MAT[;COL_EQ] , (V ×[2] MAT[;COL_INEQ])
    , (~REF_I) ,[1.5] ROW
    A preparatory sort, ensure lexicographic order
MAT ← MAT[V;] A sort matrix
ROW ← ROW[V] A sort member numbers

BEG_I ← 1 , v / (1 ↓[1] V) ≠ (~1 ↓[1] (V ← MAT[;COL_EQ]))
    A inequality?
GR_NR ← + \ BEG_I A dissect at equalities

:FOR COL :IN COL_INEQ
    REF_I ← NO_MATCH > ROW A mark references
    :SELECT REL[COL] A build next sorting vector
    :CASE '<='
        V ← SPEC_SORT (( GR_NR , MAT[;COL] , (~REF_I)
            ,[1.5] ROW ) REF_I 'Up' )
    :CASE '<'
        V ← SPEC_SORT (( GR_NR , MAT[;COL] , REF_I
            ,[1.5] ROW ) REF_I 'Up' )
    :CASE '≥'
        V ← SPEC_SORT (((-GR_NR) , MAT[;COL] , REF_I
            ,[1.5] (-ROW)) REF_I 'Down')
    :CASE '>'
        V ← SPEC_SORT (((-GR_NR) , MAT[;COL] , (~REF_I)

```

```

        , [1.5] (-ROW)) REF_I 'Down')
:ENDSELECT
MAT ← MAT[V;] ⌘ sort matrix
ROW ← ROW[V] ⌘ sort member numbers
:ENDFOR

:IF 0 < ρCOL_INEQ
  I ← v / (1 ↓ [1] V) ≠ (¬1 ↓ [1] (V ← MAT[;COL_INEQ]))
  ⌘ inequality?
  REF_I ← NO_MATCH > ROW ⌘ mark references
  BEG_I ← BEG_I v (1 , (I ~ 1 ↓ REF_I)
    v ((1 ↓ REF_I) ~ (¬1 ↓ ~ REF_I)))
    ⌘ inequalities at reference only,
    equalities if after data and at reference
:ENDIF

DAT_I ← NO_MATCH ≤ ROW ⌘ mark data

V ← BEG_I / ι 1 ↑ ρMAT ⌘ begin of each group
MAT_NR ← (1 ↓ V , (1 + 1 ↑ ρMAT)) - V ⌘ members per group

IND[(1 - NO_MATCH) + DAT_I / ROW]
  ← DAT_I / MAT_NR / NO_MATCH [ BEG_I / ROW
    ⌘ member number of first match in reference

```

6.2 Correctness of algorithm

6.2.1 Prerequisites for correctness

The algorithm completely separates equalities and inequalities. The first are trivial. We can therefore assume that we deal only with inequalities. The algorithm begins with a lexicographic sort (using ROW to ensure that at the end we get the desired first matches). It is necessary to keep the lexicographic order of the references upright until the end.

The end part is also easy. Assuming that

- each reference comes before all data points that belong in the same dissection and
- references are in lexicographic order

the data points are *immediately* preceded by the corresponding reference.

We have analyzed the building of dissections on inequalities in subsection 5.1. There we used an additional condition on data points followed by references in the case of strict inequalities.

One must observe that this is superfluous but not harmful in the case of non-strict inequalities: then such sequences never contain identical members. Therefore it can be used for all inequalities (as done in the proposed algorithm).

Let us further assume that SPEC_SORT

- sorts by exchanging consecutive pairs or members and

- respects flags by never exchanging a pair of members if the second one is flagged.

Those conditions are conservative and can be relaxed a bit. They are meant to use the references as a “scaffold” or in other words

- keep the reference members in a stable position relative to each other and
- allow data points to slide up past references as far as necessary but not more or the other way round.

We (that fore) flag the reference points. It must be noted that this “special sort” produces an order that does not depend only on the members but also on their starting order.

Further it must be noted, that for *every* $d \in \mathcal{D}$ all $r \in \mathcal{R}_{\leftrightarrow}(d)$ precede d . This however does not mean that every $r \in \mathcal{R}$ which precedes a certain d is also contained in $\mathcal{R}_{\leftrightarrow}(d)$. Such a sort does not exist in general.

As an elementary example consider $\leftrightarrow = (\leq, \leq)$, $\mathcal{D} = \{(2, 5), (4, 3)\}$ and $\mathcal{R} = \{(1, 1), (2, 4), (3, 3)\}$. We have $\mathcal{R}_{\leftrightarrow}((2, 5)) = \{(1, 1), (2, 4)\}$ and $\mathcal{R}_{\leftrightarrow}((4, 3)) = \{(1, 1), (3, 3)\}$. There is no way to sort \mathcal{M} as desired.

6.2.2 Proof of correctness

Let $d, d' \in \mathcal{D}$ and $r, r' \in \mathcal{R}$ be arbitrary but fix.

data versus reference We want to establish

$$\{r \leftrightarrow d\} \text{ or } \{\exists \tilde{r} \in \mathcal{R}, \tilde{r} \leftrightarrow d \text{ and } \tilde{r} \text{ ends up after } r \text{ and before } d\}$$

$$\Leftrightarrow r \text{ ends up before } d.$$

We implicitly use the restriction about consecutive pairs on the sort algorithm when we directly compare r with d .

1. We first want to prove that if $r \leftrightarrow d$ holds then r ends up before d .
If $r \leftrightarrow d$ holds then $r \leftrightarrow_{lex} d$ also holds. That fore r starts *before* d because of the preliminary sort.
In each step $r^{(k)} \leftrightarrow^{(k)} d^{(k)}$ is true. Should be checked (should the members be compared), at strict inequalities this determines that r stays before d . At non-strict inequalities the use of the variable REF_I as part of the sort ensures that this is the case even if $r^{(k)} = d^{(k)}$ holds. It follows that r stays before d . Note that especially for $r \in \mathcal{R}^{wk,loc}(d)$ the relation $r \leftrightarrow d$ holds per definition.
2. Next let r be such that $r \leftrightarrow_{lex} d$ does *not* hold. Because of the nature of lexicographic sort $d \leftrightarrow_{lex} r$ holds and r starts *after* d . The condition on flags ensures that this stays so.
3. Next we select a r such that $r \leftrightarrow d$ does *not* hold but $r \leftrightarrow_{lex} d$ does. r starts before d because of the preliminary sort. Let k be minimal with $r^{(k)} \leftrightarrow^{(k)} d^{(k)}$ false. Up to the $(k - 1)$ -th sort r stays before d . Let us assume that $r^{(k)}$ is compared with $d^{(k)}$.
If $\leftrightarrow^{(k)}$ is a non-strict inequality, then $r^{(k)} \neq d^{(k)}$ and the corresponding sort exchanges the pair (this is allowed as d is *not* flagged). If $\leftrightarrow^{(k)}$ is strict then the use of REF_I ensures the switch.

After that *no* switch can occur because r is flagged. This means that r ends up after d .

4. Finally if in the previous setting $r^{(k)}$ is *not* compared with $d^{(k)}$ this means that another member $\tilde{r} \in \mathcal{R}$ blocks the comparison. But $d \leftrightarrow_{lex} \tilde{r}$ cannot hold, as we have shown, and for the combination $\tilde{r} \leftrightarrow_{lex} d$ true but $\tilde{r} \leftrightarrow d$ false we can repeat the same argument.

So $\tilde{r} \leftrightarrow d$ must hold and we have found the desired intermediate reference member.

connectedness of $\mathcal{R}^{wk,loc}(d)$ We now want to show that all members of each subgroup $\mathcal{R}^{wk,loc}(d)$ end up in consecutive (uninterrupted) order. We assume $\mathcal{R}^{wk,loc}(d) \cap \mathcal{R}^{wk,loc}(d') = \emptyset$, $r \in \mathcal{R}^{wk,loc}(d)$ and $r \neq r'$.

1. We first note that $r \leftrightarrow_{lex} r'$ is equivalent to r starting before r' because of the preliminary sort. Reference points are however flagged and never exchanged. That fore

$$r \leftrightarrow_{lex} r' \iff r \text{ ends up before } r'$$

also holds.

2. Now let $r \leftrightarrow_{lex} r'$ hold. Per assumption $r \leftrightarrow d$ also holds, so r ends up before r' and d . Let r' end up before d . Then, as we have established, $r' \leftrightarrow d$ holds (or at least we can replace it without loss of generality with an appropriate \tilde{r}).

Let k be the first coordinate with $r^{(k)} \leftrightarrow (r')^{(k)}$. Because $r \neq r'$ we may assume that this is a strict inequality. But this means that $r^{(k)}$ is *not* extremal with respect to d , in contradiction to $r \in \mathcal{R}^{wk,loc}(d)$. It follows that r , d and r' end up in this order.

3. If $r' \leftrightarrow_{lex} r$ holds then of course r' ends up before r and therefore d .
4. We finally examine d' . If $\mathcal{R}^{wk,loc}(d') = \emptyset$ is empty, then d' must end up before *all* members of \mathcal{R} and therefore before r . If it is not then we may without loss of generality assume that $r' \in \mathcal{R}^{wk,loc}(d')$. Then d' ends up after r' , which ends up after d .

We have proved that our algorithm dissects \mathcal{M} correctly. So the only think that remains is to create SPEC_SORT!

6.3 Special sort algorithm with scaffold

We present some algorithms that can be used for the special sort required to effectively find weak local matches.

6.3.1 Basic facts about sort algorithms

We consider a primitive algorithm (lets call it “Primitive Sort”), Bubble Sort, Merge Sort and Natural Merge Sort (based on <https://de.wikipedia.org/wiki/Sortierverfahren>) as sort algorithms. The primitive algorithm is a dumb version of Bubble Sort that loops through the *whole* array until it gets stationary. Furthermore all those algorithms are stable.

It is clear that Primitive Sort and Bubble Sort fulfil the requirements laid out in subsection 6.2. It is also clear that (and how) they can be modified to respect the condition on flags.

Merge Sort splits the data into smaller lists and repeatedly zips together consecutive lists. Natural Merge Sort is the same except for the definition of the lists. So both algorithms violate the strict criteria (about comparing consecutive pairs of members) we laid out in subsection 6.1.

6.3.2 A false special sort algorithm

We want to show that a modification of Merge Sort can be used as a special sort. To better illustrate the problem we introduce a *false* start. Let us consider the following adaption to (Natural) Merge Sort:

1. We start with two lists, S with the starting and E with the ending members, and an empty result.
2. As long as both lists are not empty and the first member of E is *not* flagged we zip members from the lists together as in Merge Sort and append them to the result.
3. Then we append the rest of S to the result.
4. Then we append the rest of E to the result.

This preserves the stability of the relative position of flagged members and the property of not-flagged members to slide up past flagged ones but not the other way round. The non-flagged members can however slide to “high”.

To see this consider the lists $S = \{0, 1, 3, 0, 1, 3\}$ and $E = \{2, 4\}$. The first and third as well as the second members respectively are flagged. The first member of E lands at the third position of the result.

In the setting that interests us however the members of S belong to two subgroups defined by a previous step of the dissection. The obtained result destroys the dissection because it groups the data point together with a reference that does not have the right (extremal) value at a coordinate already processed.

6.3.3 A modification of (Natural) Merge Sort as special sort

Instead we propose the following modified (Natural) Merge Sort:

1. We start with two lists, S with the starting and E with the ending members, and an empty result.
2. As long as both lists are not empty and E contains a flagged member we append the result to the *last* member of E .
3. As long as both lists are not empty we zip trailing members from the lists together as in Merge Sort and append the result to them. In particular we reverse the comparison of two members in a way that preserves stability.
4. Then we append the result to the rest of S .
5. Then we append the result to the rest of E .

It is clear that the “inverse zip” as such does not alter the results of Merge Sort.

To establish assertions about the results of our special sort algorithm it is always enough to consider one member in \mathbf{S} and one in \mathbf{E} at some step of the process. In all other cases the relative positions of members are preserved.

Because the end of \mathbf{E} , beginning with the first flagged member of \mathbf{E} , lands at the end and the rest is compared normally the sort

- preserves the relative position of flagged members,
- allows non-flagged members to slide up past flagged ones but not the other way round and
- ensures that the position of two members relative to each other is only changed after comparing them.

6.3.4 Modified proof of correctness

In particular the sort does not (in contrast to the normal Merge Sort) assume that each list is “correctly” sorted. Using this we can slightly modify and generalize the proof of correctness given in subsection 6.2 (check the latter for further details).

We write the full proof down again, although in an abbreviated form. Point 1 of “data versus reference” is (only) formally modified; point 3 and 4 use the property that members are only switched around after comparison; the rest is the same.

Let $d, d' \in \mathcal{D}$ and $r, r' \in \mathcal{R}$ be again arbitrary but fix.

data versus reference We want to establish

$$\{r \leftrightarrow d\} \text{ or } \{\exists \tilde{r} \in \mathcal{R}, \tilde{r} \leftrightarrow d \text{ and } \tilde{r} \text{ ends up after } r \text{ and before } d\}$$

$$\Leftrightarrow r \text{ ends up before } d.$$

1. We first assume $r \leftrightarrow d$ holds and prove that r ends up before d . Then r starts *before* d and in each step $r^{(k)} \leftrightarrow^{(k)} d^{(k)}$ holds.

The critical step comes when $r^{(k)}$ is a member of \mathbf{S} and $d^{(k)}$ one of \mathbf{E} . But this means at worst a normal comparison of values, the order is preserved.

2. Next let r be such that $r \leftrightarrow_{lex} d$ does *not* hold. Then r starts *after* d and the condition on flags ensures that this stays so.
3. Next we select a r such that $r \leftrightarrow d$ does *not* hold but $r \leftrightarrow_{lex} d$ does. Let k be minimal with $r^{(k)} \leftrightarrow^{(k)} d^{(k)}$ false and let us assume that $r^{(k)}$ is compared with $d^{(k)}$.

We consider the step where $r^{(k)}$ is a member of \mathbf{S} and $d^{(k)}$ one of \mathbf{E} . A comparison means a normal one, as d is not flagged. The two members are switched.

After that *no* switch can occur because r is flagged. This means that r ends up after d .

4. Finally if in the previous setting $r^{(k)}$ is *not* compared with $d^{(k)}$ this means that another member $\tilde{r} \in \mathcal{R}$ blocks the comparison. Then $\tilde{r} \leftrightarrow d$ must hold and we have found the desired intermediate reference member.

connectedness of $\mathcal{R}^{wk,loc}(d)$ We now want to show that all members of each subgroup $\mathcal{R}^{wk,loc}(d)$ end up in consecutive (uninterrupted) order. We assume $\mathcal{R}^{wk,loc}(d) \cap \mathcal{R}^{wk,loc}(d') = \emptyset$, $r \in \mathcal{R}^{wk,loc}(d)$ and $r \neq r'$.

1. We first note that $r \leftrightarrow_{lex} r'$ is equivalent to r starting before r' and that reference points are never exchanged. That fore

$$r \leftrightarrow_{lex} r' \iff r \text{ ends up before } r'$$

holds.

2. Now let $r \leftrightarrow_{lex} r'$ hold. Then r ends up before r' and d . Let r' end up before d so that it may be assumed that $r' \leftrightarrow d$ holds.

Let k be the first coordinate with $r^{(k)} \leftrightarrow (r')^{(k)}$. Because $r \neq r'$ it follows that $r^{(k)}$ is *not* extremal with respect to d , in contradiction to $r \in \mathcal{R}^{wk,loc}(d)$. It follows that r , d and r' end up in this order.

3. If $r' \leftrightarrow_{lex} r$ holds then of course r' ends up before r and therefore d .
4. We finally examine d' . If $\mathcal{R}^{wk,loc}(d') = \emptyset$ is empty, then d' must end up before r . If not then we may assume $r' \in \mathcal{R}^{wk,loc}(d')$ and d' ends up after r' , which ends up after d .

We have proved that our algorithm dissects \mathcal{M} correctly. So it can be used as SPEC_SORT.

6.4 Implementation of the special sort with scaffold

We present some implementations of the special sort required to effectively find weak local matches in APL. It seems however that a “correct” and smooth solution should involve some enhancement (like an external function) on the side of the vendor.

6.4.1 Special sort in APL

The special sort can of course be implemented in APL itself. We give an example. It uses Primitive Sort. Its runtime behavior is bad regardless of the implementation.

Under APL runtime is abysmal. Still the algorithm guarantees that it only checks and exchanges consecutive pairs of members (and that it is stable). It can be used as a very transparent reference point.

Algorithm “special sort in APL”

```

DATA_V ← ⍒ 1 ↓ (SORT_V ← ⍒ 1 ↑ ρDATA)
SIGN    ← ('Up' 'Down' ⍒ cSORT) ⊃ 1 ⍒ 1
SORT_I ← 1

:WHILE SORT_I ≠ another loop?
    SORT_I ← 0
    :FOR NR :IN DATA_V ≠ work through all pairs
        NR ← 0 1 + NR ≠ pair numbers
        :IF    ~ (⍒ 1 ↑ NR) ⊃ FLAG_I
    
```

```

      A special condition:
      closing flagged members are not exchanged
:ANDIF (SIGN × (>/[1] DATA[NR;]) ι 1)
      < (SIGN × (</[1] DATA[NR;]) ι 1)
      A normal, stable sorting
      SORT_I ← 1 A another loop necessary
      DATA [NR;] ← DATA [V ← ϕNR;]
      FLAG_I[NR ] ← FLAG_I[V      ]
      SORT_V[NR ] ← SORT_V[V      ]
:ENDIF
:ENDFOR
:ENDWHILE

```

6.4.2 Special sort in .Net

It is of course also possible to create a special sort in .Net, for example in C#. The author tried out Primitive Sort, Bubble Sort, Merge Sort and Natural Merge Sort and modified each to accept the flags as required.

It was rather difficult for an APL programmer to handle simple algorithms whose code run over whole pages. Still .Net solutions that contain the algorithms were created and can be provided.

Their runtime behaviour on simple 4 byte integer data was comparable to the version 13 APL+Win Grade Up primitive \mathbb{A} . In the case of real numbers the “correct” interaction with comparison tolerance was not even researched.

The algorithms were used by APL+Win in the form of a dll. Probably they could be bound in Dyalog directly as C# code. What the author (as an absolute .Net novice) could not provide was an overload which appears as *one* method in APL+Win and that can handle different data types and ranks without increasing runtime by 1–3 orders of magnitude...

We present the core of the modified Merge Sort as an example for a possible implementation under .Net.

Algorithm “special sort in .Net”

```

index = len - 1;
// Sort data by merging 2 established consecutive runs at a time.
while (index > (increment - 1))
{
    i_r = index;
    i_l = index - increment;
    begin_r = i_l + 1;
    begin_l = begin_r - increment;
    if (begin_l < 0)
    {
        begin_l = 0;
    }
    i_z = i_r;

    // Put trailing elements of righth run into result,
    // starting with first flagged element.

```

```

if (run_flagged[i_r])
{
    end_r = begin_r;
    while (flags_unsorted[end_r] == false)
    {
        end_r = end_r + 1;
    }
    end_r = end_r - 1;
    while (i_r > end_r)
    {
        data_sorted[i_z] = data_unsorted[i_r];
        flags_sorted[i_z] = flags_unsorted[i_r];
        if (create_vector)
            { vector_sorted[i_z] = vector_unsorted[i_r]; }
        i_r = i_r - 1;
        i_z = i_z - 1;
    }
}
run_flagged[index] =
    (run_flagged[index - increment] | run_flagged[index]);

if (sort_up) { i_comp_l = i_l; i_comp_r = i_r; }
else { i_comp_l = i_r; i_comp_r = i_l; }

// Put last element of left run into result if it is
// (strictly) greater than the last element of the right run,
// else use the latter.
while ((i_l >= begin_l) && (i_r >= begin_r))
{
    if (data_unsorted[i_comp_l] > data_unsorted[i_comp_r])
    {
        data_sorted[i_z] = data_unsorted[i_l];
        flags_sorted[i_z] = flags_unsorted[i_l];
        if (create_vector)
            { vector_sorted[i_z] = vector_unsorted[i_l]; }
        i_l = i_l - 1;
        if (sort_up) { i_comp_l = i_comp_l - 1; }
        else { i_comp_r = i_comp_r - 1; }
    }
    else
    {
        data_sorted[i_z] = data_unsorted[i_r];
        flags_sorted[i_z] = flags_unsorted[i_r];
        if (create_vector)
            { vector_sorted[i_z] = vector_unsorted[i_r]; }
        i_r = i_r - 1;
        if (sort_up) { i_comp_r = i_comp_r - 1; }
        else { i_comp_l = i_comp_l - 1; }
    }
    i_z = i_z - 1;
}

```

```

}
// Put residual elements of left run into result.
while (i_l >= begin_l)
{
    data_sorted[i_z] = data_unsorted[i_l];
    flags_sorted[i_z] = flags_unsorted[i_l];
    if (create_vector)
        { vector_sorted[i_z] = vector_unsorted[i_l]; }
    i_l = i_l - 1;
    i_z = i_z - 1;
}
// Put residual elements of righth run into result.
while (i_r >= begin_r)
{
    data_sorted[i_z] = data_unsorted[i_r];
    flags_sorted[i_z] = flags_unsorted[i_r];
    if (create_vector)
        { vector_sorted[i_z] = vector_unsorted[i_r]; }
    i_r = i_r - 1;
    i_z = i_z - 1;
}
index = index - (2 * increment);
}

```

6.5 A sorting algorithm for weak global matches

6.5.1 Outline of a possible algorithm

We now search for weak local matches. As in subsection 5.1 the variables REF and DAT represent \mathcal{R} and \mathcal{D} respectively and REL stands for the relation \leftrightarrow . The remarks about \mathbb{Z} , \mathbb{R} and alphanumeric data hold.

We want to use the alternate approach to strong global matches presented in subsection 5.2 to find the desired matches. We can find weak matches using the algorithm "weak local match" and then decide if those are also (weak) global (matches). To these means we start with an arbitrary but fix $d \in \mathcal{D}$ and denote with r the first weak local match (if it exists.)

We indicate the potential for a weak global match as $\mathbb{1}_0^{wk,gl}(d)$. For each coordinate we successively examine if $r^{(k)}$ is also globally extremal. If not then there exists no match. In formulas

$$\mathbb{1}_k^{wk,gl}(d) = \begin{cases} 1 & \text{if } k = 0 \text{ and } \mathcal{R}^{wk,loc}(d) \neq \emptyset \\ \mathbb{1}_{k-1}^{wk,gl}(d) & \text{if } k \geq 1 \text{ and} \\ & r^{(k)} = \text{extr}\{(r')^{(k)}; r' \in \mathcal{R}_{\leftrightarrow}(d)\} \\ 0 & \text{else} \end{cases}$$

If and only if $\mathbb{1}_n^{wk,gl}(d)$ is true there exists a weak global match ($\mathcal{R}^{wk,gl}(d) \neq \emptyset$) and (then) r is the first match. (We used a particular order to check the conditions of extremality but it is obviously arbitrary and not relevant for the end result.)

6.5.2 Outline of implementation of trivial case in APL

We try to find an implementation of the proposed algorithm suitable for APL. The problem is the restriction of the extremal values to $\mathcal{R}_{\leftrightarrow}(d)$. We have seen in subsection 4.4 that it is not (in general) possible to sort \mathcal{M} in such a way that *each* d is preceded *exactly* by $\mathcal{R}_{\leftrightarrow}(d)$.

If $\mathcal{D} = \{d\}$ contains only one element it is easy to see a solution. Successively for each coordinate in their natural order

1. drop all members following d and then
2. sort using Grade Up or Grade Down.

This uses the fact that $\mathcal{R}_{\leftrightarrow}(d)$ precedes d at the end of the algorithm "weak local match". Unfortunately $\mathcal{R}_{\leftrightarrow}(d)$ depends on d in a way that does not allow for easy generalisation.

Of course the problem can be solved by looping through the data points d . These however will not yield a practically useful algorithm. More generally the solution will at best behave like $\mathcal{O}(M^2 \cdot \log(M))$ with $M = |\mathcal{M}|$.

6.5.3 Discussion of the problem

We cannot of course use a normal sort to find the relevant extremal value for an arbitrary coordinate. The sort cannot distinguish between $\mathcal{R}_{\leftrightarrow}(d)$ and the rest of \mathcal{R} . In that sense it is too flexible.

On the other side the special sort with scaffold is too rigid. Using it on the result of "weak local match" leaves \mathcal{M} completely stationary. More general it is easy to see that a reference can block a data point, not allowing it to reach another reference with higher value that precedes the first.

It is possible to define an intermediate sort variation that

- compares adjacent pairs m and m' ,
- exchanges a leading reference m and/or a trailing data point m' if $m \leftrightarrow m'$ does not hold (ordinary sort condition) and
- never exchanges a leading data point with a trailing reference.

This new special asymmetric sort algorithm comes near to a solution but is not good enough. To pinpoint the problem consider first the case of one coordinate. Let d be a data point preceded by a reference r that happens to be the relevant extremal one.

It can happen that there is a data point d' with $r \leftrightarrow d'$ and a reference r' with $r' \leftrightarrow d$. Let further $r' \leftrightarrow r$ hold. If the 4 members start in the order $(\dots, r, d', r', d, \dots)$ then they are stationary under the new algorithm, so the right extremal value cannot be matched to d .

As an example let $d = (2, 3)$, $d' = (1, 4)$, $r = (1, 3)$ and $r' = (2, 2)$ be 4 members with 2 coordinates. The result of "weak local match" leaves \mathcal{M} in the order (r, d', r', d) . Although $\mathcal{R}^{wk,loc}(d) = \{r'\}$ and $\mathcal{R}^{wk,gl}(d) = \emptyset$ evidently hold sorting the second coordinate encounters the problem just described.

Furthermore $\mathcal{R}^{wk,gl}(d') = \{r\}$ holds and the example shows that there is no obvious way to "sort" the 4 members the "right way" with respect to the second coordinate. The desired result is (r, d, d', r') and there does not seem to be a rule that would lead to the pair (r', d) getting exchanged.

6.5.4 A possibly suboptimal circumvention

To give any algorithm for weak global matches at all we use the symmetry of their definition. Both $\mathcal{R}_{\leftrightarrow}(d)$ and the extremal values defining $\mathcal{R}^{wk,gl}(d)$ do not depend on the order of the coordinates.

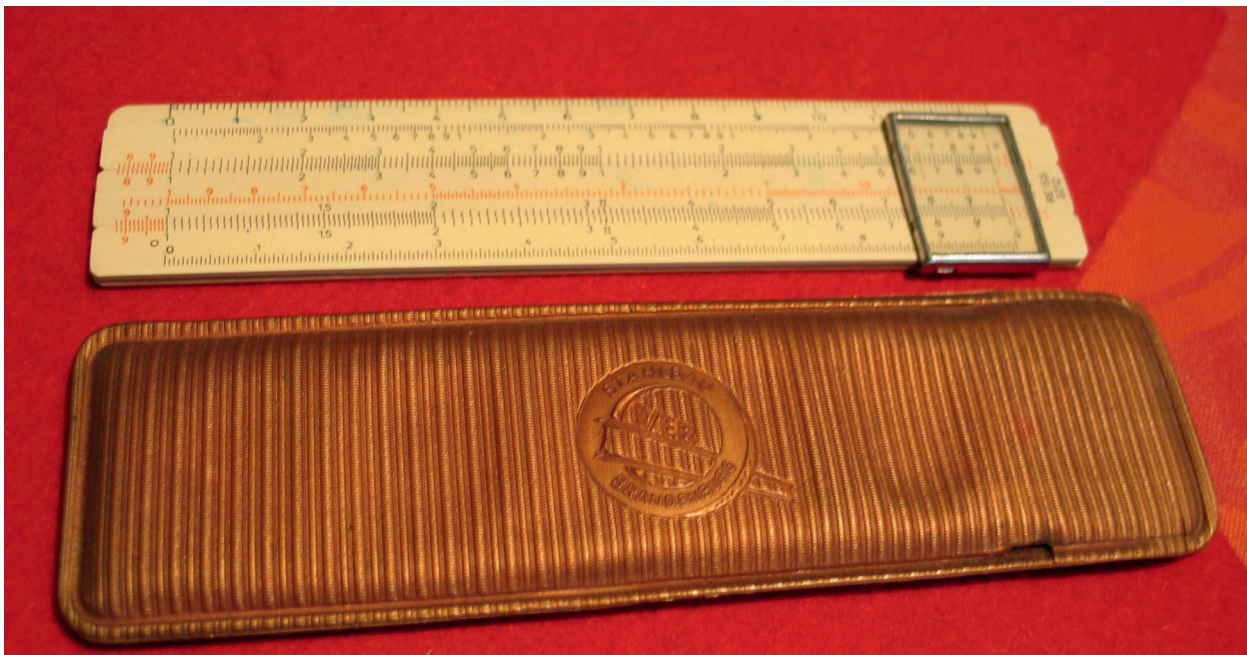
For each $d \in \mathcal{D}$ and for each permutation $\pi \in S_n$ we can define a weak local match $\mathcal{R}_{\pi}^{wk,loc}(d)$. If there is a weak global match then $\mathcal{R}^{wk,gl}(d) = \mathcal{R}_{\pi}^{wk,loc}(d)$ must hold for every π .

On the other hand the *first coordinate* of each weak local match is a *global* extremal value with respect to d . For the existence of a weak global match a reference must be found whose coordinates are all extremal with respect to d (and that fore $\mathcal{R}_{\leftrightarrow}(d)$). If all $\mathcal{R}_{\pi}^{wk,loc}(d)$ are *identical* they definitely fulfil the requirement.

Furthermore it is clearly enough to examine the n cyclic permutations $\pi \in S_n$ of length n because each of them makes another coordinate the first one. We may that fore circumvent the sorting problem:

1. for each $\pi \in S_n$ cyclic of length n find $\mathcal{R}_{\pi}^{wk,loc}(d)$ and note the index of the first match for all $d \in \mathcal{D}$
2. compare the n indices for each d
3. if and only if all indices are identical and less or equal than $|\mathcal{R}|$ (see subsection 4.2 on the convention for non-existent matches) for a specific d there is a weak global match $\mathcal{R}^{wk,gl}(d)$

This algorithm can obviously be implemented in APL as a loop on weak local matches. It is not however obvious if there exists a significantly better solution.



KALAHA

Erfahrungen bei der Implementation von neuronalen Netzen in APL
Dipl.Math. Ralf Herminghaus, Dezember 2018

Die Schlagzeile

Es begann eigentlich alles recht zufällig mit einer Schlagzeile in einer einschlägigen wissenschaftlichen Zeitschrift:



Der Firma Google ist es gelungen, mit Hilfe künstlicher neuronaler Netze den amtierenden (menschlichen) GO-Weltmeister zu entthronen. Die Schlagzeilen waren vielversprechend: Soweit dem Artikel zu entnehmen war, hatte das neuronale Netz nicht nur effiziente Strategien entwickelt, um eine GO-Partie gegen einen menschlichen Spieler zu gewinnen – auch die Regeln des Spiels hatte das Netz allein anhand von Beispiel-Partien selbst erschlossen. Insgesamt zeigte der Artikel auf eindrückliche Weise, wozu künstliche neuronale Netze heutzutage in der Lage sind.

Der Gedanke liegt nahe, so etwas einmal selbst auszuprobieren.

Nun bietet sich APL als Programmiersprache für eine derartige Programmierung geradezu an. Der Algorithmus zur Implementation eines neuronalen Netzes lässt sich in APL in wenigen Zeilen hinschreiben.

```
[0] Z←KALA_NN_CALCULATE RA;NN;EING;INP;I;OUTP
[1] A-----
[2] A RH,08.03.2018 NN-Kalkulation durchführen
[3] A
[4] A Die Aktivität eines neuronalen Netzes ist ja nichts
[5] A anderes als eine Sequenz von Matrix-Multiplikationen
[6] A mit jeweils einer nachgeschalteten Schwellwert-Funktion.
[7] A
[8] A-----
[9] (NN EING)←RA
[10]
[11] A-----
[12] A Relevant für die Entscheidung sind nur 12 Felder, die
[13] A die Spiel-Situation charakterisieren
[14] A-----
[15] INP←εEING,1
[16]
[17] A---Schleife über alle Schichten des Netzes
[18] :for I :in itpNN
[19] A-----
[20] A Matrix-Multiplikation und nachfolgend
[21] A Tangens-Hyperbolicus
[22] A-----
[23] INP←?o{(I=NN)+.×INP)
[24] A---Schwellwert-Dimension hinzufügen
[25] INP←εINP,1
[26] :endfor
[27]
[28] A-----
[29] A Ausgabe separieren
[30] A-----
[31] OUTP←~1+INP
[32]
[33] Z←OUTP
[34]
```

Wo also steckt das Problem?

Rechenleistung

Das Problem liegt in der verfügbaren Rechenleistung der verwendeten Hardware:

Alpha-Zero: 11,5000000000 PetaFlops
Mein Rechner: 0,00000009321 PetaFlops

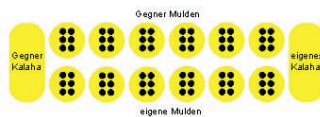
Google hat für den verwendeten Algorithmus ALPHA-GO schlichtweg deutlich mehr Rechenpower zur Verfügung als ich auf meinem kleinen Laptop. Dieser Umstand bewog mich dann dazu, es erst einmal mit einem etwas einfacheren Spiel zu versuchen: KALAHA

KALAHA

KALAHA ist ein traditionelles afrikanisches Brettspiel mit sehr einfachen Regeln und nur wenigen möglichen Alternativen für jeden Zug. Trotzdem ist das Spiel nicht ganz trivial – ideal für Experimente am Computer.

Kalaha

Spielregeln



Startaufstellung

je 6 Kugeln werden in die 12 kleinen Mulden gelegt

Gewinner ist, wer bei Spielende die meisten Kugeln in seinem Kalaha hat.

Wer am Zuge ist, leert eine seiner Mulden und verteilt die Kugeln, jeweils eine, reihum im Gegenuhrzeigersinn in die nachfolgenden Mulden. Dabei wird auch das eigene Kalaha gefüllt. Das Gegner Kalaha wird ausgelassen.

Fällt die letzte Kugel ins eigene Kalaha, ist der Spieler nochmals am Zuge.

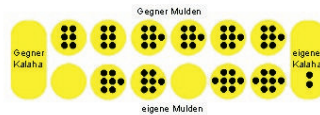
Fällt die letzte Kugel in eine leere Mulde auf der eigenen Seite, wird diese Kugel und alle Kugeln in der Gegner Mulde gegenüber, ins eigene Kalaha gelegt und der Gegner hat den nächsten Zug.

Das Spiel ist beendet, wenn alle Mulden eines Spielers leer sind. Der Gegner bekommt dann alle Kugeln aus seinen Mulden in sein Kalaha.

Beispiel:



1. Spieler hat die Mulde ganz links geleert und ist noch mal am Zuge, da die letzte Kugel in seinem Kalaha gelandet ist.



1. Spieler hat seine 4. Mulde geleert, jetzt ist der 2. Spieler am Zuge.

Die einfachen Regeln ermöglichen es, das Spiel mit relativ wenig Aufwand auf einem Computer zu implementieren. Andererseits wird durch die geringe Anzahl von Alternativen pro Zug auch die Analyse von Spiel-Situationen weniger rechenintensiv.

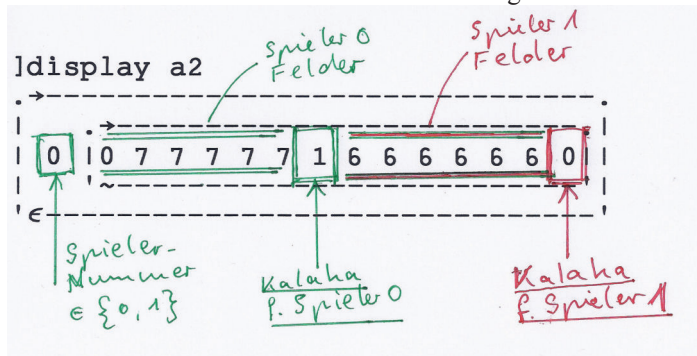
Zum Vergleich:

GO: ca. 350 Wahlmöglichkeiten pro Zug
Schach: ca. 30-40 Wahlmöglichkeiten pro Zug
KALAHA: max. 6 Wahlmöglichkeiten pro Zug

Grundlagen der Implementation von KALAHA in APL

Das erste Problem bei der Implementation eines Spiels auf dem Computer ist es, eine geeignete interne Darstellung der möglichen Spiel-Situationen zu finden. Ist diese Darstellung ungünstig gewählt, so folgen daraus fast unweigerlich hohe Aufwände bei der Analyse von Spiel-Situationen und damit insgesamt eine geringe Spielstärke des resultierenden Spiele-Algorithmus.

Für APL bot sich nach etwas Nachdenken die folgende interne Darstellung an:



Eine Spiel-Situation wird abgebildet durch einen Vektor mit zwei Elementen:

- 1) Eine Zahl (0 oder 1), die angibt, welcher Spieler gerade am Zug ist
- 2) Einen Vektor aus 14 Zahlen, die die aktuelle Belegung der 14 Mulden im KALAH-Brett repräsentieren

Für die praktische Anwendung des Spiele-Programms benötigt man allerdings noch eine besser lesbare Darstellung des intern abgespeicherten Zustandes nach außen. Der Einfachheit halber ist dies als Übertragung in eine Character-Matrix realisiert.

```

KALA_SHOW a2
=====
      16 6 6 6 6 6 6 |
0 |           | 1
      10 7 7 7 7 7 |
Sp0=====
    
```

Hier wird eine Spiel-Situation a2 mit Hilfe der Funktion KALA_SHOW aus der internen Darstellung in eine besser lesbare Darstellung als 5x19-Character-Matrix überführt. Andere (schönere) Darstellungen wären möglich, hätten aber auch deutlich mehr Aufwand bedeutet.

Einige KALAH-Spiel-Züge als Beispiel

Beispielhaft sollen hier nun einige Züge aus der Anfangsphase einer KALAH-Partie durchgespielt werden. Im ersten Schritt wird mit Hilfe der niladischen Funktion KALA_START die Anfangs-Belegung des Spiels erzeugt.

Danach werden mit Hilfe der dyadischen Funktion KALA_ZUG mehrere Züge durchgeführt. Dabei bezeichnet das linke Argument jeweils eine Eingabe-Situation und das rechte Argument bezeichnet den durchgeführten Zug. Da es im KALAH-Spiel für einen Spieler immer nur die Möglichkeit gibt, aus einer von maximal 6 Mulden auszuwählen, kann das rechte Argument sinnvoll nur die Werte {1,...,6} annehmen.

Im letzten Schritt werden mit Hilfe der Funktion KALA_INVERT die Rollen der Spieler „0“ und „1“ gegeneinander vertauscht. In der Realität entspricht dies etwa dem Umdrehen des Spielbrettes.


```

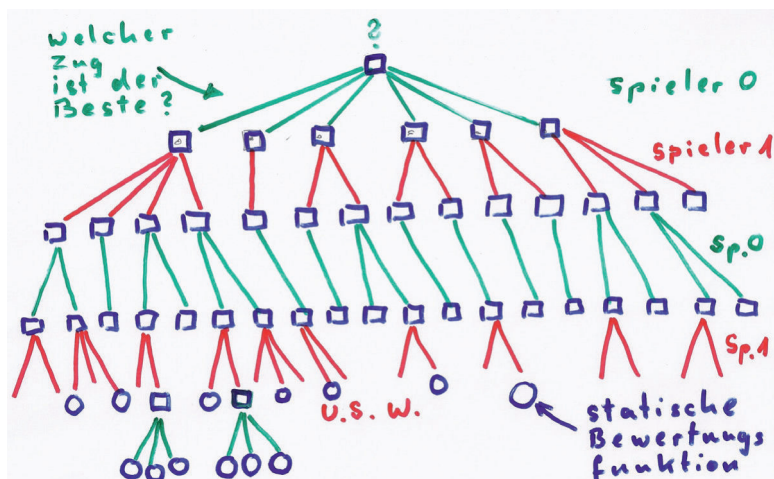
jdisplay KALA_START
-----
| 0 | 6 6 6 6 6 6 0 6 6 6 6 6 0 |
|-----|
a1←KALA_START
a2←a1 KALA_ZUG 1
jdisplay a2
-----
| 0 | 10 7 7 7 7 7 1 6 6 6 6 6 0 |
|-----|
a3←a2 KALA_ZUG 2
jdisplay a3
-----
| 1 | 10 0 8 8 8 8 2 7 7 6 6 6 0 |
|-----|
a4←a3 KALA_ZUG 5
jdisplay a4
-----
| 0 | 11 1 9 9 8 8 2 7 7 6 6 0 7 1 |
|-----|
a5←a4 KALA_ZUG 1
jdisplay a5
-----
| 1 | 10 2 9 9 8 8 2 7 7 6 6 0 7 1 |
|-----|
a6←KALA_INVERT a5
jdisplay a6
-----
| 0 | 17 7 6 6 0 7 1 10 2 9 9 8 8 2 |
|-----|
KALA_SHOW a1
-----
| 16 6 6 6 6 6 |
0 | | 0
| 16 6 6 6 6 6 |
Sp0=====
KALA_SHOW a2
-----
| 16 6 6 6 6 6 |
0 | | 1
| 10 7 7 7 7 7 |
Sp0=====
KALA_SHOW a3
-----
| 16 6 6 6 7 7 |
0 | | 2
| 10 0 8 8 8 8 |
Sp1=====
KALA_SHOW a4
-----
| 17 0 6 6 7 7 |
1 | | 2
| 11 1 9 9 8 8 |
Sp0=====
KALA_SHOW a5
-----
| 17 0 6 6 7 7 |
1 | | 2
| 10 2 9 9 8 8 |
Sp1=====
KALA_SHOW a6
-----
| 18 8 9 9 2 0 |
2 | | 1
| 17 7 6 6 0 7 |
Sp0=====
    
```

Programmierung von 2-Personen-Strategiespielen im Allgemeinen

Das Vorgehen bei der Programmierung von 2-Personen-Strategiespielen ist generell immer das gleiche, egal ob es sich nun um SCHACH, DAME, MÜHLE, GO oder eben KALAH handelt.

Um als Spieler zu entscheiden, welcher Zug in einer gegebenen Spiel-Situation denn der günstigste ist, untersucht man jeden einzelnen der möglichen (hier z.B. 6) Züge und bewertet die daraus resultierenden Gewinn-Aussichten. Dazu muss man jeweils betrachten, welche Zugmöglichkeiten sich aus der eigenen Entscheidung denn für den Gegner ergeben würden, jeweils unter Berücksichtigung der Möglichkeiten, die sich aus der Entscheidung des Gegners für einen selbst ergeben haben, usw. usw. ...

Realisiert werden derartige Bewertungen als rekursive Funktionen, die einen Baumgraphen der unterschiedlichen möglichen Spielsituationen untersuchen. Ideal wäre es natürlich, wenn man den gesamten Baum mit allen möglichen Spielsituationen untersuchen könnte. Die ist aber naturgemäß nur bei sehr wenigen (trivialen) Spielen wie etwa TIC-TAC-TOE möglich.

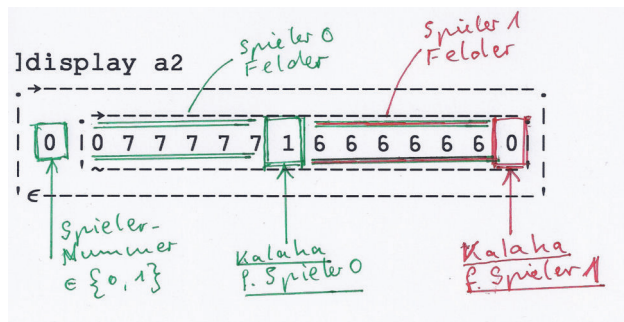


Bei nicht-trivialen Spielen kann eine solche Suche naturgemäß nur in sehr begrenzter Tiefe vorgenommen werden. Aus diesem Grund wird die Baumsuche in der Regel ab einer gewissen Suchbaum-Tiefe abgebrochen und die Situation an den Blättern des Baumes über eine sogenannte „statische Bewertungsfunktion“ eingeschätzt. Diese Funktion ist dann nicht mehr rekursiv. Oft ist sie sogar sehr einfach definiert. So könnte es evtl. ausreichen, einfach die Differenz aus der erreichten Anzahl der eigenen Gewinnpunkte und der Gewinnpunkte des Gegners zur Bewertung zu verwenden.

Im Schach zum Beispiel bewertet man eine Situation gerne über die Anzahl der eigenen Figuren minus der Anzahl der Figuren des Gegners. Dabei fließt dann in der Regel auch noch der „Wert“ der jeweiligen Figur und ihre aktuelle Position auf dem Spielfeld mit ein. Eine Dame z.B. ist mehr wert als ein Springer, und der wiederum mehr als ein Bauer, und generell wiegen Figuren in der Mitte des Spielfeldes schwerer als die am Rand. Die statische Bewertungsfunktion kann ganz unterschiedlich definiert und evtl. auch relativ kompliziert definiert sein - das hängt von der Phantasie und den Fähigkeiten des jeweiligen Programmierers ab. Wichtig ist: Die statische Bewertungsfunktion ist nicht rekursiv und bezüglich der verwendeten CPU-Zeit immer vergleichsweise „billig“.

Statische Bewertungsfunktion beim Spiel KALAHA

Beim Spiel KALAHA etwa ist die „natürlichste“ statische Bewertungsfunktion die Differenz der Inhalte der beiden Kalaha-Felder:



$$BEW(a_2) = TAB[7] - TAB[14]$$

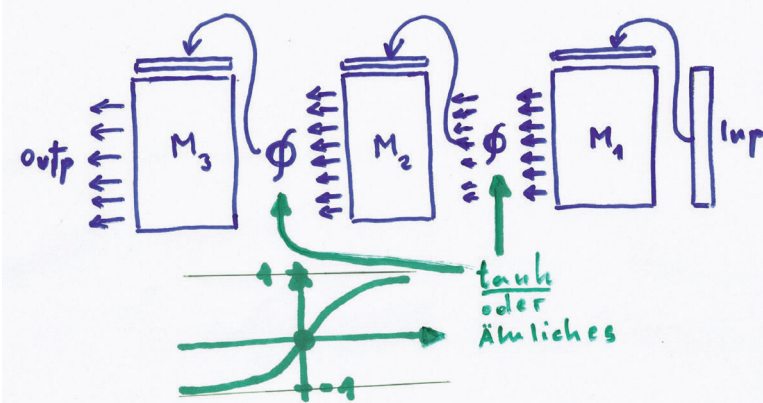
Diese Funktion ist vor allen Dingen sehr einfach definiert und erfüllt durchaus die Erwartungen an eine statische Bewertungsfunktion. Am Ende des Spiels gibt sie etwa genau wieder, welcher der beiden Spieler wie hoch gewonnen hat. Zudem ist intuitiv einsehbar, dass es im Verlauf des Spiels von Vorteil sein dürfte, möglichst viele Gewinnpunkte in der eigenen Kalaha angesammelt zu haben, auch wenn das Spiel noch nicht zu Ende ist. Es sind aber durchaus auch andere statische Bewertungsfunktionen denkbar, die die Situation differenzierter bewerten. So könnten z.B. die Belegungen der übrigen Felder auf dem KALAHA-Brett jeweils mit einer gewissen Gewichtung in die Bewertung einer Spiel-Situation mit einfließen. Der Phantasie sind hier keine Grenzen gesetzt.

Einsatz neuronaler Netze als statische Bewertungsfunktion

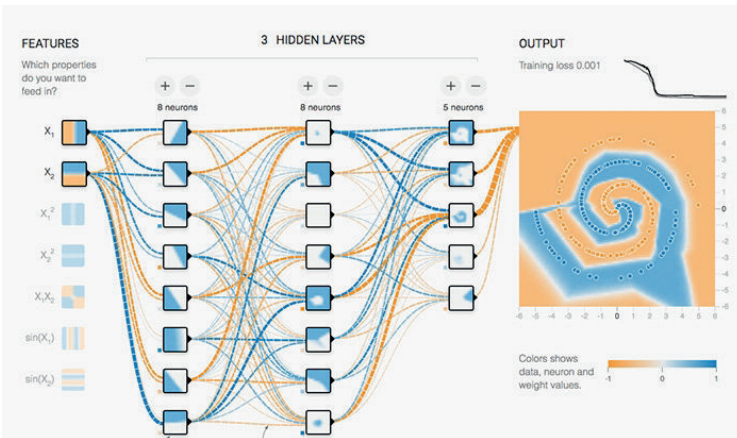
Mathematisch betrachtet stellen neuronale Netze eine sehr flexibel einsetzbare Klasse von Funktionen dar, die durch geeignete Parametrisierung fast beliebige funktionale Abhängigkeiten abbilden können. Im vorliegenden Fall soll nun ein neuronales Netz darauf hin trainiert werden, die bisher verwendete statische Funktion BEW zu ersetzen. Dabei wäre es natürlich von Vorteil, wenn das neuronale Netz die Ergebnisse einer (teuren!) Baumsuche möglichst genau simulieren könnte - natürlich ohne dafür den entsprechend hohen Aufwand an CPU-Kapazität zu benötigen.

Wie funktioniert ein „neuronales Netz“ ?

Ein einfaches „neuronales Netz“ mit n Schichten lässt sich realisieren als Sequenz von n Matrix-Multiplikationen mit eingeschobenen Schwellwert-Funktionen (Tangens -Hyperbolicus, Sigmoid-Funktion, o.Ä.) . Es gibt noch andere Typen von neuronalen Netzen – die wir hier aber nicht betrachten wollen.



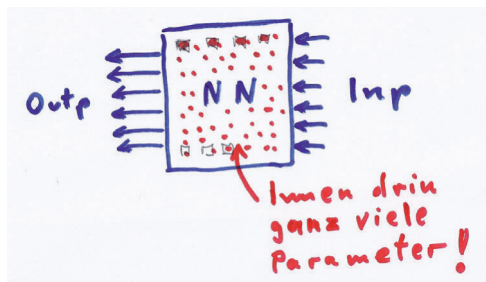
Eine andere übliche Darstellungsweise gibt eher den Netz-Charakter einer solchen Funktion wieder, meint aber im Grunde das Gleiche:



In APL sieht die Implementation eines solchen mehrschichtigen neuronalen Netzes dann folgendermaßen aus:

```
[0] Z←KALA_NN_CALCULATE RA;NN;EING;INP;I;OUTP
[1] -----
[2] A RH,08.03.2018 NN-Kalkulation durchführen
[3] A
[4] A Die Aktivität eines neuronalen Netzes ist ja nichts
[5] A anderes als eine Sequenz von Matrix-Multiplikationen
[6] A mit jeweils einer nachgeschalteten Schwellwert-Funktion.
[7] A
[8] A -----
[9] (NN EING)←RA
[10]
[11] A -----
[12] A Relevant für die Entscheidung sind nur 12 Felder, die
[13] A die Spiel-Situation charakterisieren
[14] A -----
[15] INP←εEING,1
[16]
[17] A---Schleife über alle Schichten des Netzes---
[18] :for I :in ttpNN
[19] A -----
[20] A Matrix-Multiplikation und nachfolgend
[21] A Tangens-Hyperbolicus
[22] A -----
[23] INP←7σ((I←NN)+.xINP)
[24] A---Schwellwert-Dimension hinzufügen---
[25] INP←εINP,1
[26] :endfor
[27]
[28] A -----
[29] A Ausgabe separieren
[30] A -----
[31] OUTP←~1↑INP
[32]
[33] Z←OUTP
[34]
```

GANZ grob vereinfacht kann man es auch so sehen:

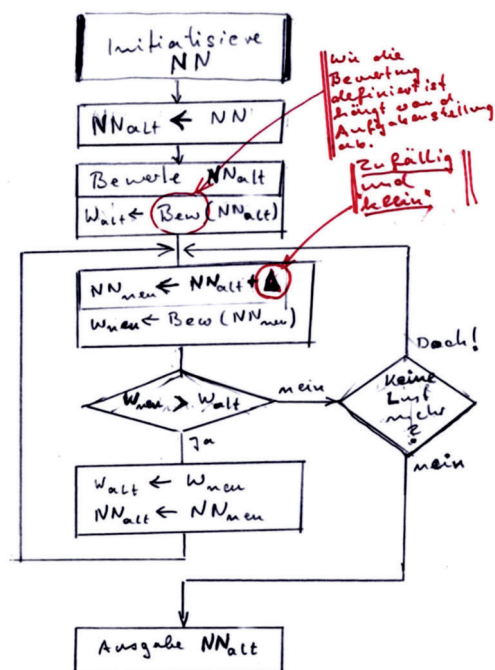


Das „Training“ eines neuronalen Netzes erfolgt einfach, indem die Parameter (d.h. die Koeffizienten der beteiligten Matrizen) solange manipuliert werden, bis die gewünschte Zuordnung von Input zu Output mit befriedigender Genauigkeit erreicht ist. Wenn das Netz eine große Anzahl von Beispiel-Fällen korrekt bearbeitet, kann man darauf hoffen, dass es das auch bei anderen - bisher noch nicht aufgetretenen Fällen – ebenso korrekt tut.

Trainingsmethoden für neuronale Netze

Nun ist das Training eines solchen neuronalen Netzes alles andere als trivial. Eine Menge unterschiedlicher Methoden wurde entwickelt, um mit möglichst wenig Rechenaufwand geeignete Parameter zu finden.

Um das Prinzip deutlich zu machen, sei hier ein sehr einfacher Evolutions-Algorithmus dargestellt, der über kurz oder lang die Parameter eines neuronalen Netzes optimieren kann.



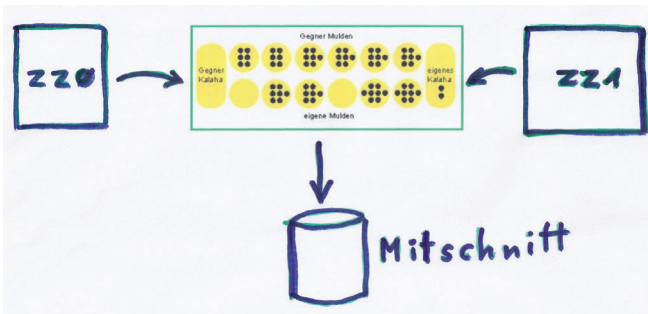
Der hier dargestellte Evolutions-Algorithmus ist extrem einfach und – leider – auch sehr ineffizient. D.h. das Training des Netzes braucht viel Zeit. Andere Trainings-Verfahren wie z.B. „Back-Propagation“ sind deutlich effizienter, aber dafür nicht in allen Fällen anwendbar. Diese Verfahren finden eine optimale Parameter-Belegung in der Regel über Algorithmen, die ein modifiziertes Newton-Verfahren oder einen Steepest-Descend-Ansatz implementieren. Welcher Ansatz im konkreten Fall der richtige ist, hängt in vielen Fällen von der Problemstellung ab.

Die Strategie, die ich zum Training der Bewertungsfunktion angewendet habe war etwa die folgende:

- 1) Erzeuge zunächst eine große Menge an realistischen Spiel-Situationen (etwa 10000 Stück)
- 2) Bewerte jede dieser Situationen mit Hilfe eines rekursiven Baumsuch-Verfahrens und speichere die Ergebnisse in einer Liste ab. (Vorsicht! Das kann lange dauern, weil die rekursive Bewertungsfunktion sehr aufwändig ist.)
- 3) Trainiere ein neuronales Netz anhand der vorliegenden Liste darauf, die rekursive Bewertungsfunktion möglichst gut zu imitieren. (Vorsicht! Auch das kann lange dauern – vor allem mit dem oben beschriebenen einfachen Trainings-Verfahren.)
- 4) Ersetze die einfache Bewertungs-Funktion (TAB[7] – TAB[14]) durch das nun fertig trainierte neuronale Netz
- 5) Herauskommen sollte dabei eine deutlich spielstärkere Version des bisherigen Algorithmus

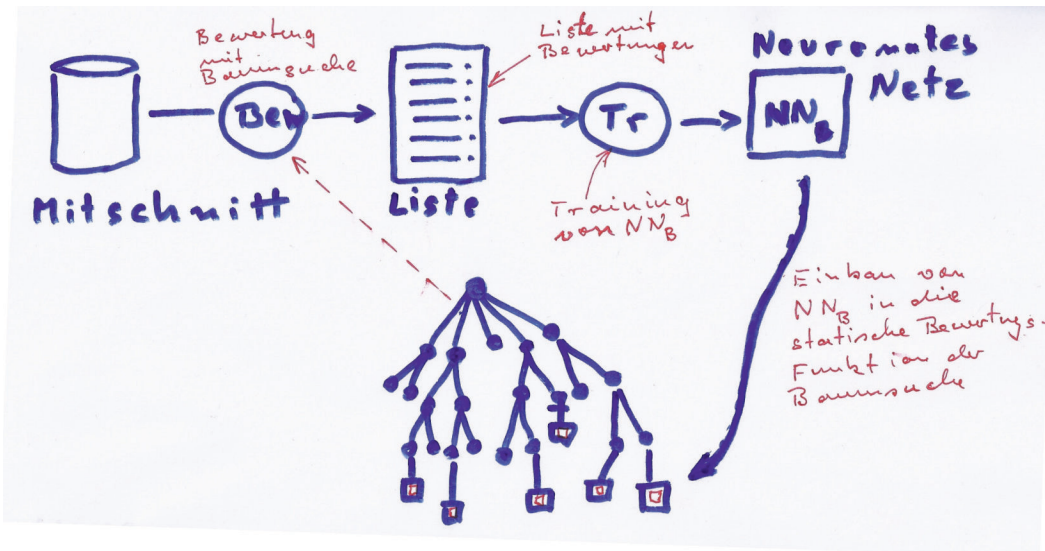
Schritt 1:

Zunächst wird eine Liste mit 10000 realistischen Spiel-Situationen erzeugt, indem man zwei Zufalls-Zahlen-Generatoren gegeneinander spielen lässt (dass die nicht besonders gut spielen, ist erst einmal egal)



Schritt 2:

Nun wird jede Spielsituation der Liste konventionell mit Hilfe eines Baumsuch-Algorithmus bewertet und das Ergebnis der Bewertung in die Liste eingetragen



Schritt 3:

Danach wird ein neuronales Netz darauf trainiert, die Bewertungen der Liste möglichst gut zu imitieren.

Schritt 4:

Nach dem Training wird die statische Bewertungsfunktion des Baumsuch-Algorithmus durch das neuronale Netz ersetzt

Schritt 5:

Das Resultat dieser Aktion ist wieder ein Baumsuch-Algorithmus, allerdings mit einer statischen Bewertungsfunktion, die deutlich leistungsfähiger ist als die simple Differenz von Gewinnpunkten.

Einige Programm-Auszüge

Leider lässt sich das Programm aus Platzgründen hier nicht vollständig wiedergeben. Um aber einen Eindruck zu vermitteln, wie bei der Programmierung vorgegangen wurde, hier einige ausgewählte Passagen:

```
[0] Z←KALA_START
[1] A-----
[2] A RH,07.06.2017 Startkonfiguration des KALHA-Spiels
[3] A-----
[4] Z←0 (14p(6 6 6 6 6 6 0))
[5]

[0] Z←KALA_INVERT KONFIG;SPIELER;TAB
[1] A-----
[2] A RH,12.02.2018 Rollen-Wechsel der Spieler
[3] A Die Spielsituation wird hier "invertiert"
[4] A Spieler 0 und Spieler 1 tauschen die Rollen
[5] A-----
[6] (SPIELER TAB)←KONFIG
[7] TAB←7φTAB
[8] :if (SPIELER=1)
[9]     SPIELER←0
[10] :elseif (SPIELER=0)
[11]     SPIELER←1
[12] :endif
[13]
[14] Z←(SPIELER TAB)

[0] Z←KALA_GAME_OVER KONFIG;SPIELER;TAB;GEW1;GEW2
[1] A-----
[2] A RH,11.06.2017 Prüfen, ob das Spiel zuende ist
[3] A-----
[4] A-----
[5] (SPIELER TAB)←KONFIG
[6] Z←(∧/0=TAB[1 2 3 4 5 6])∨(∧/0=TAB[8 9 10 11 12 13])∨(SPIELER=2)
```

Hier zum Beispiel die rekursiv definierte Bewertungsfunktion. Neben der zu bewertenden Spiel-Konfiguration (KONFIG) wird die Tiefe der Baumsuche als linkes Argument mitgegeben.

```
[0] Z←TIEFE KALA_BEWERTUNG KONFIG;SPIELER;TAB;GEW;LISTE_ZUEGE;
[1] A-----
[2] A RH,03.03.2018 Eine Konfiguration bewerten
[3] A-----
[4] (SPIELER TAB)←KONFIG
[5]
[6] A-----
[7] A Bewertung bei Spiel-Ende
[8] A In diesem Fall ist die Bewertung trivial
[9] A-----
[10] :if (∧/0=TAB[16])∨(∧/0=TAB[7+16])
[11]     Z←(+/TAB[17])-(+/TAB[7+17])
[12]     →0
[13] :endif
[14]
[15] A-----
[16] A Wenn als Evaluations-Tiefe Null angegeben ist,
[17] A wird die Konfiguration nur noch STATISCH bewertet -
[18] A d.h. mit einem nicht rekursiven "einfachen"
[19] A Algorithmus
[20] A-----
[21] :if (TIEFE=0)
[22]     Z←KALA_STATIC_BEWERTUNG KONFIG
[23]     →0
[24] :endif
[25]
[26] A-----
[27] A Liste der Nachfolge-Konfigurationen erzeugen
[28] A Wichtig dabei ist, daß dabei auch alle Konfigurationen
[29] A berücksichtigt sind, die durch Mehrfach-Ziehen
[30] A entstehen könnten.
[31] A-----
[32] LISTE_NACHFOLG←KALA_ALLE_NACHF KONFIG
[33] LISTE_BEW←(TIEFE-1) KALA_BEWERTUNG LISTE_NACHFOLG
[34]
[35] A-----
[36] A Ist der Spieler 1 am Zug, dann wird er sicherlich
[37] A die Alternative mit der geringsten Bewertung wählen.
[38] A Umgekehrt wird Spieler 0 die Alternative mit der
[39] A höchsten Bewertung wählen.
[40] A-----
[41] :if (SPIELER=1)
[42]     Z←[/εLISTE_BEW
[43] :elseif (SPIELER=0)
[44]     Z←[/εLISTE_BEW
[45] :endif
[46] →0
```

Und hier eine statische Bewertungsfunktion, die als Kern ein neuronales Netz NN verwendet:

```
[0] Z←KALA_NN_STATIC_BEWERTUNG RA;NN;KONFIG;SPIELER;TAB;REVER
[1] A-----
[2] A RH.03.03.2018 Statische Bewertung
[3] A mit Hilfe eines neuronalen Netzes
[4] A-----
[5] A-----
[6] (NN KONFIG)←RA
[7] (SPIELER TAB)←KONFIG
[8] REVERS_BEW←(SPIELER=1)
[9]
[10] A-----
[11] A Da das Netz nur auf Situationen trainiert ist,
[12] A bei denen Spieler0 am Zug ist, muß die Konfiguration
[13] A evtl. zuvor invertiert werden
[14] A-----
[15] :if REVERS_BEW
[16] KONFIG←KALA_INVERT KONFIG
[17] (SPIELER TAB)←KONFIG
[18] :endif
[19]
[20] A---Die Kalahas zuerst leeren-----
[21] (KO K1)←TAB[7 14]
[22] INP←TAB[(114)~(7 14)]
[23]
[24] A---Dann die Bewertung durchführen-----
[25] WERT←10×+/KALA_NN_CALCULATE NN INP
[26] WERT←WERT+KO-K1
[27]
[28] :if REVERS_BEW
[29] WERT←-WERT
[30] :endif
[31] Z←WERT
[32]
[33] →0
```

Mit der hier vorgestellten Strategie war es dann letztendlich möglich, ein KALAHA-Programm zu realisieren, das durchaus in der Lage ist, einen (wenig versierten) menschlichen Spieler zu schlagen. Insofern kann man von einem Erfolg sprechen.

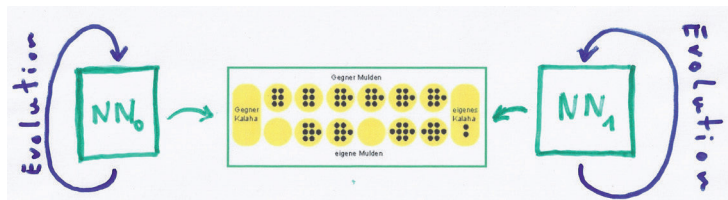
Einschränkend muss aber bemerkt werden, dass die oft behaupteten „intellektuellen Fähigkeiten“ von neuronalen Netzen in diesem Fall nicht deutlich spürbar wurden. Ein Programm mit einer ganz simplen statischen Bewertungsfunktion (Differenz der beiden Kalahas) zeigte - zumindest gefühlt - eine ähnliche Spielstärke wie ein Programm, das die statische Bewertungsfunktion über ein neuronales Netz realisiert hatte.

Unsupervised Learning

„Unsupervised Learning“ beschreibt eine Trainings-Strategie, bei der neuronale Netze allein daran gemessen werden, ob sie eine gegebene Aufgabe erfolgreich bewältigen. Eine Liste mit INPUT-OUTPUT-Vorgaben, die das Netz möglichst gut simulieren soll, wird dabei nicht vorgegeben. Meine Versuche, ein neuronales Netz durch „unsupervised learning“ für das Spiel KALAHA zu trainieren, waren leider nicht wirklich von Erfolg gekrönt.

Abschließen möchte ich den Artikel daher mit einer Reihe von – leider fehlgeschlagenen - Strategieansätzen, deren Grundidee sich erst einmal sehr überzeugend anhört, die aber letztendlich einfach nicht funktionieren.

Versuch 1 : Co-Evolution zweier neuronaler Netze

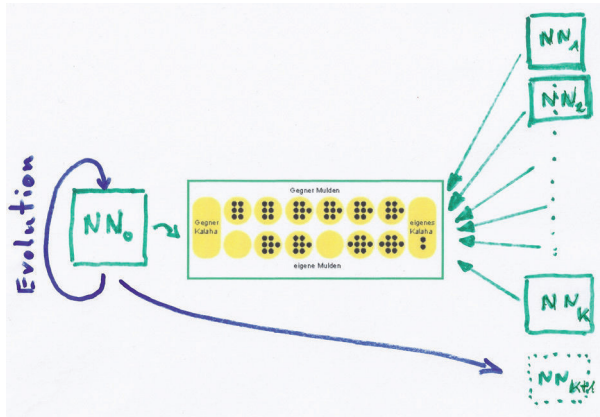


Die Idee dahinter:

Man lässt zwei neuronale Netze sehr oft gegeneinander spielen und versucht, die Spiel-Stärke durch eine Art Evolutionsprozess immer weiter zu steigern. Zunächst wird NN₀ durch Evolution solange optimiert, bis es NN₁ schlagen kann. Danach werden die Rollen getauscht und NN₁ wird solange optimiert, bis es NN₀ schlägt, dann werden die Rollen wieder getauscht, usw. Aus der Konkurrenz der beiden Netze sollten sich im Laufe des Evolutionsprozesses immer höhere Spielstärken ergeben, die letztendlich zu einem wirklich starken Spiel-Algorithmus führen ... Hört sich gut an – funktioniert aber nicht

Bei Experimenten mit dieser Strategie gewinnt man fast den Eindruck, als würden die beiden beteiligten Netze nicht wirklich gegeneinander konkurrieren, sondern sich eher auf einem sehr niedrigen Spielstärke-Niveau miteinander einigen. Die Bezeichnung „Gentlemen-Agreement“ trifft die Sache vielleicht nicht ganz – drängt sich aber auf. Am Schluss des Trainingsprozesses spielen beide Gegner jedenfalls wirklich schlecht ...

Versuch 2: Verschärfte Evolutions-Bedingungen



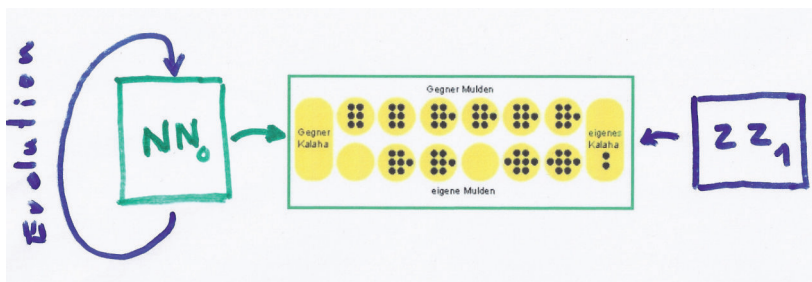
Die Idee dahinter:

Man lässt ein Netz NN_0 gegen eine ganze Liste von Gegnern spielen und etabliert einen entsprechenden Evolutionsmechanismus. Wenn NN_0 alle seine Gegner schlagen kann, kommt NN_0 selbst auf die Liste der Gegner und das Spiel geht mit einem neuen „Herausforderer“ von neuem los. Insgesamt muss sich also jeder neue „Herausforderer“ gegen die ganze Liste vorheriger „Gewinner“ behaupten, etwa nach dem Motto: „Weltmeister wird nur, wer alle vorherigen Weltmeister besiegen kann.“ Insgesamt wird sich so die Spiel-Stärke des jeweiligen „Gewinners“ immer weiter erhöhen.

Hört sich auch gut an - funktioniert aber auch nicht.

Ab einer gewissen Länge der Liste ist der „Herausforderer“ nicht mehr in der Lage, gegen alle Gegner auf der Liste zu gewinnen. Der Optimierungsprozess bleibt auf einem gewissen, sehr niedrigen Level einfach stecken.

Versuch 3: Spiel gegen einen Zufallszahlen-Generator



Die Idee dahinter:

NN_0 wird darauf trainiert, gegen einen Zufallszahlen-Generator eine möglichst hohe Quote von Spielen zu gewinnen. Wenn NN_0 zuverlässig gegen einen Zufallszahlen-Generator gewinnen kann, gewinnt es evtl. auch zuverlässig gegen einen menschlichen Gegner ... (?)

(funktioniert aber auch nicht)

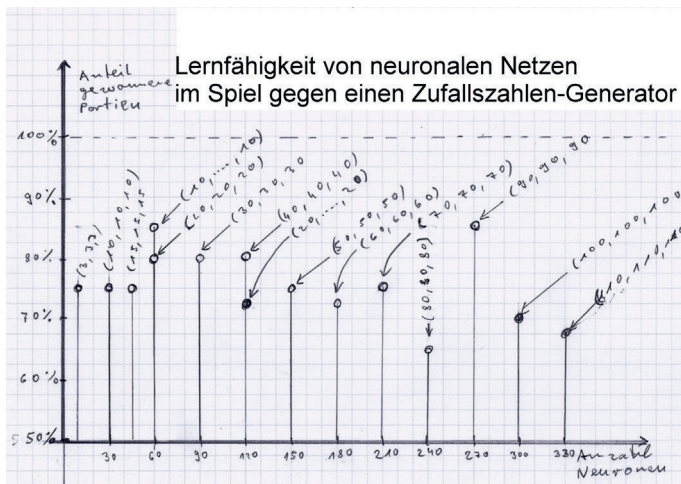
Ein Zufallszahlen-Generator ist per definitionem der schwächste denkbare Gegner in diesem Spiel. Trotzdem war es in den Experimenten nicht möglich, ein neuronales Netz soweit zu trainieren, dass es einen Zufallszahlen-Generator zuverlässig schlagen konnte. Die maximal erreichbare Gewinn-Quote blieb bei maximal etwa 85% der Spiele stecken, und zwar weitgehend unabhängig von der Größe des eingesetzten neuronalen Netzes.

Mögliche Ursache der Mißerfolge

Bei der Frage nach der Ursache dieser Misserfolge drängt sich ein möglicher Grund für das Versagen neuronaler Netze förmlich auf: Die mangelnde Größe der untersuchten Netze.

Während Google Zehntausende von Prozessoren für die Analyse eines GO-Spieles bereitstellen kann, ist dies auf dem eigenen PC schlichtweg nicht möglich. Man könnte sich daher das Versagen der oben beschriebenen Ansätze mit der zu geringen Anzahl der zur Verfügung stehenden Neuronen erklären. Bei genauerer Untersuchung lässt sich dieses Argument aber nicht halten.

Um den Einfluss der Netzgröße auf die Spielstärke einigermaßen objektiv messen zu können, habe ich in einem längeren Experiment neuronale Netze unterschiedlicher Dimensionen gegen einen Zufallszahlen-Generator trainiert. Das Ergebnis war einigermaßen ernüchternd: Die Spielstärke neuronaler Netze gegen einen Zufallszahlen-Generator blieb weitgehend unabhängig von der vorgegebenen Neuronenanzahl und der Anzahl der neuronalen Verbindungen.



Keines der getesteten Netze erreichte eine Gewinn-Quote von mehr als 85%.

Auch waren die erreichten Spielstärken weitgehend unabhängig von der Anzahl der verwendeten Neuronen.

Ergo: Es ist nicht so einfach, wie es sich in den Veröffentlichungen immer anhört ...

Über neuronale Netze und ihre phänomenalen Fähigkeiten wird gerade in der jüngsten Vergangenheit viel publiziert. Nach einigen Jahrzehnten Pause ist das Thema „Künstliche Intelligenz“ wieder in Mode gekommen. Gesichtserkennung, Spracherkennung, autonomes Fahren, intelligente Roboter, usw. usw. Alles ist möglich und die Aussichten sind einfach phantastisch!!!

Schwieriger wird es, wenn man versucht, diese phänomenalen Möglichkeiten der neuen Technik einmal im konkreten Experiment selbst nachzuvollziehen. In diesem Fall stößt man schnell an Grenzen. Dass ein künstliches neuronales Netz unter gewissen Umständen gut auf gewisse Reaktionen hin „trainiert“ werden kann, kann man an vielen Stellen nachlesen. Aber was tun, wenn das Training nicht gelingen will? Hierüber sagen die zahlreichen Publikationen dann leider meist nichts.

Hardware-Entwicklung

Neuronale Netze gibt es bereits seit Mitte der 60-er Jahre, aber außer einigen Nischen-Anwendungen hatten sie in der Vergangenheit keine nennenswerte Bedeutung, da die notwendige Rechnerkapazität in der Regel nicht verfügbar war oder unbezahlbar blieb.

Mit der Nutzung von Grafik-Prozessoren (GPU) als billige Alternative zu Super-Rechnern hat sich dies in den vergangenen 10 Jahren drastisch geändert. Heute stehen für Spezialanwendungen Rechenkapazitäten zur Verfügung, die früher undenkbar waren. Viele Anwendungen lassen sich nun mit neuronalen Netzen realisieren, die zuvor unrealistisch teuer gewesen wären.

Mit der Bereitstellung sogenannter „Tensor-Processing-Units“ (TPU) geht GOOGLE nun noch einen Schritt weiter. Dieser Prozessor-Typ ist schwerpunktmäßig allein auf die Durchführung von Matrixmultiplikationen hin konstruiert und kann so die Funktionalität neuronaler Netze mit bisher unerreichter Effizienz abbilden. Als billige Alternative zur GPU könnte dieser Prozessortyp daher in Zukunft noch viel von sich reden machen.

Und: Weitere Verbesserungen der Performance sind zu erwarten ...

Ausblick

Wie ein neuronales Netz im Prinzip funktioniert, weiß man seit langer Zeit. Wie man ein solches Netz am sinnvollsten trainieren kann, ist aber immer noch eher eine Kunst als eine Wissenschaft. Leistungsfähigere Hardware und neue Typen von neuronalen Netzen sind in den vergangenen Jahrzehnten entwickelt worden. Sie können Aufgaben lösen, für die die einfachen Perceptrons der 60-er Jahre nicht geeignet waren. Mit der Verfügbarkeit kostengünstiger Hardware wird sich die Bandbreite von Anwendungen, die auf der Technik neuronaler Netze basieren, in Zukunft deutlich vergrößern. Zu welchen Leistungen diese Mechanismen dann in der Lage sein werden, ist heute noch ziemlich unklar. Eine Reihe von spektakulären Erfolgen lässt aber Einiges erwarten.

Mittelfristig wird man jedenfalls nicht darum herumkommen, sich mit dieser Technologie auseinanderzusetzen.

Es bleibt spannend ☺ !



APL2: \mathbb{C} kann \mathbb{C}

Dieter Kilsch
Technische Hochschule Bingen*

Inhaltsverzeichnis

- 1 Einleitung
- 2 Ein sehr einfaches Auto-Modell
- 3 Die allgemeine homogene Lösung
- 4 Spezielle inhomogene Lösung und Lösung des Anfangswertproblems

1 Einleitung

Mit Service Level 23 von IBM APL2 (April 2016) wurde die interne Domino-Funktion \mathbb{C} auf komplexe Zahlen \mathbb{C} erweitert. Dies betrifft alle ihre Einsatzbereiche:

- Invertieren regulärer komplexer Matrizen: $\mathbb{C}A$ liefert A^{-1} .
- Berechnen der Pseudo- oder Linksinversen $A^* = (\overline{A^T A})^{-1} \cdot \overline{A^T}$ einer Matrix vollen Ranges mit mehr Zeilen als Spalten nach Moore [3] und Penrose [4], s. auch [2, Anhang 2]) durch $\mathbb{C}A$.
- Lösen quadratischer linearer Gleichungssysteme $A \cdot \vec{x} = \vec{b}$ mit regulären komplexen Koeffizientenmatrizen A durch $\mathbf{x} \leftarrow \mathbf{b} \mathbb{C}A$.
- Berechnen der ausgeglichenen Lösung eines komplexen, überbestimmten, linearen Gleichungssystems $A \cdot \vec{x} = \vec{b}$ mit einer Koeffizientenmatrix A vollen Ranges durch $\mathbf{x} \leftarrow \mathbf{b} \mathbb{C}A$. Die ausgeglichene Lösung \vec{x} minimiert $\|A\vec{x} - \vec{b}\|$.

Damit entfällt das Laden und Aufrufen der Bibliotheksfunktion DOMINO aus der Bibliothek MATHFNS und lässt die Darstellung des Lösungswegs vereinfachen. Der Weg zum Lösen eines linearen Gleichungssystems wird anhand eines einfachen Beispiels dargestellt.

2 Ein sehr einfaches Auto-Modell

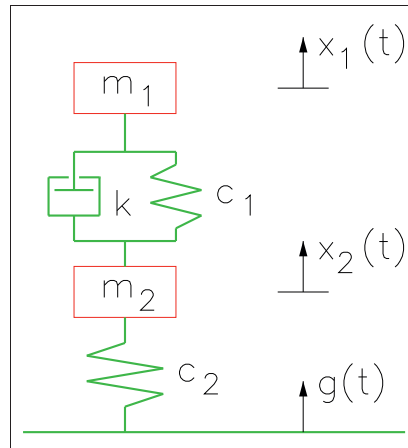
Ein sehr, sehr einfaches Modell eines Fahrzeugs, das aber zur Demonstration des Einsatzes der Domino-Funktion im ersten und dritten Fall ausreicht, besteht aus zwei vertikal gelagerten Massen, s. Abb. 1. Die untere Masse stellt das federnd über der Fahrbahn gelagerte Rad und die gegen das Rad gefedert und gedämpft gelagerte Karosserie dar. Die vertikale Auslenkung des Rads aus der Gleichgewichts- oder Ruhelage wird mit x_1 , diejenige der Karosserie mit x_2 bezeichnet. Das Beispiel ist in [1] enthalten.

Die die vertikale Auslenkung beschreibenden Differenzialgleichungen wird mit Hilfe des zweiten Newtonschen Axioms aufgestellt:

$$\begin{aligned} m_1 \ddot{x}_1 &= -c_1(x_1 - x_2) - k(\dot{x}_1 - \dot{x}_2) \\ m_2 \ddot{x}_2 &= -c_1(x_2 - x_1) - c_2(x_2 - g(t)) - k(\dot{x}_2 - \dot{x}_1) \end{aligned}$$

mit den Federkonstanten c_1 und c_2 und der Dämpfungskonstante k . Die Federkraft wird proportional zur Federauslenkung genommen

*www.th-bingen.de/person/dieter-kilsch/

Abb. 1: Vertikale Lage der Karosserie (x_1) und des Rads (x_2)

Hieraus erhält man mit $x_3 = \dot{x}_1$ und $x_4 = \dot{x}_2$ und den Zahlenwerten

$$\begin{aligned} m_1 &= 1000 \text{ kg}, & c_1 &= 40 \text{ kN/m}, & k &= 16 \text{ t/s} \\ m_2 &= 50 \text{ kg}, & c_2 &= 50 \text{ kN/m}, & g(t) &= 0.1 \text{ m für } t \geq 0 \end{aligned}$$

ein System linearer Differenzialgleichung erster Ordnung

$$\begin{aligned} \dot{\vec{x}}(t) = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \\ \ddot{x}_1 \\ \ddot{x}_2 \end{pmatrix} &= \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{c_1}{m_1} & \frac{c_1}{m_1} & -\frac{k}{m_1} & \frac{k}{m_1} \\ \frac{c_1}{m_2} & -\frac{c_1+c_2}{m_2} & \frac{dk}{m_2} & -\frac{k}{m_2} \end{pmatrix} \vec{x}(t) + \begin{pmatrix} 0 \\ 0 \\ 0 \\ g(t) \frac{c_2}{m_2} \end{pmatrix} \\ &= A \cdot \vec{x}(t) + \vec{b} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -40 & 40 & -16 & 16 \\ 800 & -1800 & 320 & -320 \end{pmatrix} \vec{x}(t) + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 100 \end{pmatrix} \\ &= A \cdot \vec{x}(t) + \vec{b}(t) \\ \vec{x}_0 &= \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}; \quad g(t) = 0.1 \text{ m} \end{aligned}$$

Die Störfunktion $f(t) = 0.1$ für $t \geq 0$ bedeutet, dass das Auto auf ein Stufe von 10 cm auffährt.

3 Die allgemeine homogene Lösung

Die Lösung des homogenen linearen Gleichungssystems $\dot{x}_h(t) = A \cdot x_h(t)$ wird mit Hilfe der Eigenwerte und Eigenvektoren der Systemmatrix A berechnet. Die Routine Charpo berechnet das hierzu benötigte charakteristische Polynom der Matrix mit Hilfe ihrer zentralen Hauptunterdeterminanten:

```
mat←4 4ρ0 0 1 0 0 0 1 ^-40 40 ^-16 16 800 ^-1800 320 ^-320
ϕpol0←Charpo mat
      40000 16000 1840 336 1
pol1←(i^-1+ρpol0)×1+pol0
```

Das charakteristische Polynom lautet damit

$$f_A(\lambda) = \lambda^4 + 336\lambda^3 + 1840\lambda^2 + 16000\lambda + 40000$$

Abb. 2 zeigt, dass das charakteristische Polynom nur zwei Nullstellen hat, in der Nähe von -320 und -3 . Diese werden mit dem Newton-Verfahren berechnet:

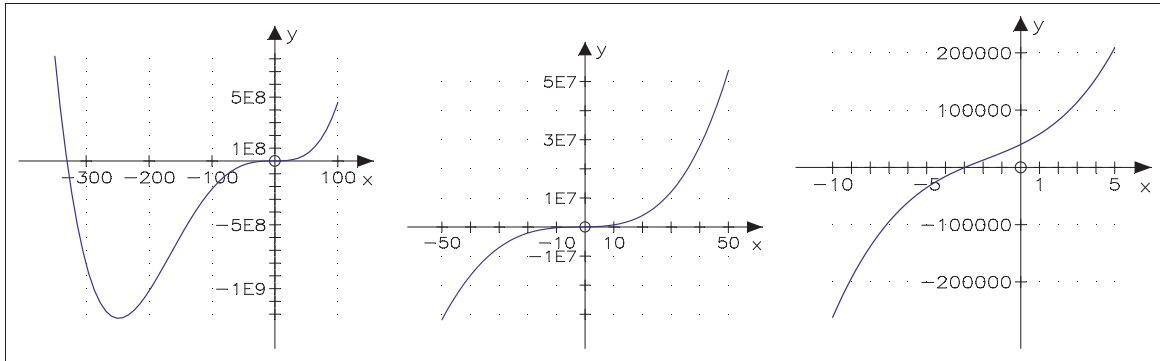


Abb. 2: Das charakteristische Polynom in den Bereiche [-350, 300], [-50, 50] [-10, 5]

```
⊞xN←2⊞⊞⊞300 ⊞100+Newton⊞⊞(c⊞t⊞ϕpol⊞),⊞⊞01⊞
⊞330.5793156 ⊞2.9689688
```

Das Horner-Verfahren erlaubt, das Restpolynom zu berechnen, dessen komplexe Nullstellen mit der pq-Formel ermittelt werden. Mit

```
,r←1+⊞xN[2]Horner 1+⊞xN[1]Horner pol0
40.75479204 2.451715606 1
,r←(⊞0.5×r[2])+1 ⊞1×((0.25×r[2]*2)-r[1])*0.5
⊞1.225857803J6.265146821 ⊞1.225857803J⊞6.265146821
lambda←xN,r
```

folgt

$$f_A(\lambda) = (\lambda + 330.57932)(\lambda + 2.96897)(\lambda^2 + 2.45172\lambda + 40.75479)$$

Die Nullstellen des letzten Faktors sind komplex und komplex konjugiert zueinander. Alle Nullstellen wurden auf lambda zusammengefasst. Dies sind die Eigenwerte der Systemmatrix, deren zugehörige Eigenvektoren mit dem Gauß-Verfahren als Lösung von

$$(A - \lambda_i \text{Id})\vec{e}_i = \vec{0}$$

berechnet werden:

```
e←2+2 Gausst A-lambda[1]×Id 4⊞E+4 1P(,3 ⊞1t⊞),⊞1⊞e
1 0 0 ⊞0.0001526336545
0 1 0 0.00302499265
0 0 1 0.05045752903
0 0 0 0
e←2+2 1 Gausst A-lambda[2]×Id 4⊞E+E,(,3 ⊞1t⊞),⊞1⊞e
1 0 0 ⊞1.927367528
0 1 0 0.3368172815
0 0 1 5.722294057
0 0 0 0
e←2+2 1 Gausst A-lambda[3]×Id 4⊞E+E,(,3 ⊞1t⊞),⊞1⊞e
1 0 0 0.1037600809J0.1530978787
0 1 0 0.03007886291J0.1537278565
0 0 1 ⊞1.086375793J0.462395912
0 0 0 0
E←E,+E[;3] a komplex konjugiert
```

Die Eigenvektoren komplex konjugierter Eigenwerte sind komplex konjugiert, daher konnte $\vec{e}_4 = \overline{\vec{e}_3}$ sofort übernommen werden. Die Eigenvektoren werden zur Fundamentalmatrix $E = [\vec{e}_1, \vec{e}_2, \vec{e}_3, \vec{e}_4]$ zusammengefasst:

```
E
⊞0.0001526336545 ⊞1.927367528 0.1037600809J0.1530978787 0.1037600809J⊞0.1530978787
0.00302499265 0.3368172815 0.03007886291J0.1537278565 0.03007886291J⊞0.1537278565
0.05045752903 5.722294057 ⊞1.086375793J0.462395912 ⊞1.086375793J⊞0.462395912
⊞1 ⊞1 ⊞1 ⊞1
```

Die allgemeine homogene Lösung der Differentialgleichung $\dot{\vec{x}}_h(t) = A \cdot \vec{x}_h(t)$ lautet somit

$$\vec{x}_h(t) = \sum_{i=1}^4 c_i \vec{e}_i e^{\lambda_i t}$$

4 Spezielle inhomogene Lösung und Lösung des Anfangswertproblems

Eine spezielle Lösung der inhomogenen Differenzialgleichung $\dot{\vec{x}}(t) = A \cdot \vec{x}(t) + \vec{b}$ erhält man mit der Idee, dass nach langer Zeit sich das Auto ausgeschaukelt hat und keine vertikale Bewegung mehr vorhanden ist, also $\dot{\vec{x}}(t) = 0$ gilt. Aus $\vec{0} = A \cdot \vec{x}_s(t) + \vec{b}$ folgt dann die konstante Lösung $\vec{x}_s = -A^{-1}\vec{b}$, die in APL berechnet wird:

```
bs←0 0 0 100
,xs←-bs⊖A
0.1 0.1 0 0
```

Die Lösung des Anfangswertproblems ist mit Hilfe der komplexen Eigenvektoren und mit der Erweiterung von \mathbb{R} auf \mathbb{C} einfach. Für $t = 0$ folgt aus $\vec{x}_0 = \vec{x}(0) = \vec{x}_s + \sum_{i=1}^4 c_i \vec{e}_i = x_s + E \cdot \vec{c}$ sofort $\vec{c} = E^{-1} \cdot (\vec{x}_0 - \vec{x}_s)$. Die Berechnung mit APL ergibt:

```
x0←0 0 0 0
,c←(x0-xS)⊖E
0.2926399848 -0.009464405199 -0.1415877898J0.290057577 -0.1415877898J-0.290057577
```

Offensichtlich sind c_3 und c_4 komplex konjugiert. Das muss immer so sein:

$$\begin{aligned} \vec{x}(t) &= x_s(t) + c_1 \vec{e}_1 e^{\lambda_1 t} + c_2 \vec{e}_2 e^{\lambda_2 t} + c_3 \vec{e}_3 e^{\lambda_3 t} + c_4 \vec{e}_4 e^{\lambda_4 t} \\ \Rightarrow \overline{\vec{x}(t)} &= \overline{x_s(t)} + \overline{c_1 \vec{e}_1 e^{\lambda_1 t}} + \overline{c_2 \vec{e}_2 e^{\lambda_2 t}} + \overline{c_3 \vec{e}_3 e^{\lambda_3 t}} + \overline{c_4 \vec{e}_4 e^{\lambda_4 t}} \\ \Rightarrow \vec{x}(t) &= x_s(t) + c_1 \vec{e}_1 e^{\lambda_1 t} + c_2 \vec{e}_2 e^{\lambda_2 t} + \overline{c_3 \vec{e}_3} e^{\lambda_3 t} + \overline{c_4 \vec{e}_4} e^{\lambda_4 t} \\ \Rightarrow c_4 &= \overline{c_3} \quad \text{und} \quad c_4 \vec{e}_4 = \overline{c_3 \vec{e}_3} . \end{aligned}$$

Mit $\lambda_i = \alpha_i + j\beta_i$ ($i = 3, 4$) folgen jetzt

$$\begin{aligned} c_3 \vec{e}_3 e^{\lambda_3 t} + c_4 \vec{e}_4 e^{\lambda_4 t} &= e^{\alpha_3 t} (c_3 \vec{e}_3 (\cos(\beta_3 t) + j \sin(\beta_3 t)) + \overline{c_3 \vec{e}_3} (\cos(\beta_3 t) - j \sin(\beta_3 t))) \\ &= e^{\alpha_3 t} (2 \operatorname{Re}(c_3 \vec{e}_3) \cos(\beta_3 t) + 2 \operatorname{Im}(c_4 \vec{e}_4) \sin(\beta_3 t)) \end{aligned}$$

Die Berechnung der Ergebnisse mit APL liefert:

```
⊖EE←1 1 2 2×[2]9 9 9 110[2]E×[2]c
-0.00004466671033 0.01824138725 -0.1181967205 -0.0168392148
0.0008852338033 -0.00318777523 -0.09769745857 0.02608277067
0.01476589053 -0.05415810962 0.03939221909 0.7611622908
-0.2926399848 0.009464405199 0.2831755796 0.5801151541
```

In mathematischer Notation lautet die Lösung

$$\begin{aligned} x(t) &= \begin{pmatrix} .1 \\ .1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -.0000447 \\ .0008852 \\ .0147660 \\ -.2926400 \end{pmatrix} e^{-330.580 t} + \begin{pmatrix} .0182414 \\ -.0031878 \\ -.0541581 \\ .0094644 \end{pmatrix} e^{-2.96897 t} \\ &\quad + e^{-1.22585 t} \left(\cos(6.26515 t) \begin{pmatrix} -.1181967 \\ -.0976975 \\ .0393922 \\ .2831756 \end{pmatrix} + \sin(6.26515 t) \begin{pmatrix} -.0168392 \\ .0260828 \\ .7611623 \\ .5801152 \end{pmatrix} \right) , \end{aligned}$$

deren erste beiden Komponenten die Bewegung der Karosserie und des Rads angeben. Abb. 3 enthält die grafische Darstellung dieser beiden Komponenten.

Der Schwingungsanteil kann durch eine Sinusfunktion mit Phasenverschiebung dargestellt werden. Aus

$$a \cos(\omega t) + b \sin(\omega t) = A \sin(\omega t + \varphi) = A \cos(\omega t) \sin(\varphi) + A \sin(\omega t) \cos(\varphi)$$

folgt durch Koeffizientenvergleich

$$a = A \sin(\varphi) \quad \text{und} \quad b = A \cos(\varphi)$$

Damit sind A und φ die Polarkoordinaten des ebenen Vektors $(b; a)$, A also dessen Betrag und φ dessen Argumentwinkel. Dies wird berechnet

```
(+/2+[2]EE*2)*.5 ♂ Amplituden
0.1193902169 0.101119258 0.7621809365 0.6455400846
(⊖EE[;3]<0)+^3⊖+/'c[2]2+[2]EE ♂ neg. Phasenwinkel
3.283108032 2.880702501 1.519089736 1.116684204
```

und eingesetzt:

$$x(t) = \begin{pmatrix} .1 \\ .1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} -.0000447 \\ .0008852 \\ .0147660 \\ -.2926400 \end{pmatrix} e^{-330.580 t} + \begin{pmatrix} .0182414 \\ -.0031878 \\ -.0541581 \\ .0094644 \end{pmatrix} e^{-2.96897 t} + e^{-1.225858 t} \begin{pmatrix} .119390 \sin(6.26515 t + 3.283108) \\ .101120 \sin(6.26515 t + 2.880703) \\ .762181 \sin(6.26515 t + 1.51909) \\ .645540 \sin(6.26515 t + 1.116684) \end{pmatrix}.$$

Die vertikale Bewegung der Karosserie wird in der ersten Komponente beschrieben:

$$x_1(t) = 0.1 - .0000447 e^{-330.580 t} + .0182414 e^{-2.96897 t} - 0.119390 e^{-1.225858 t} \sin(6.26515 t + 3.283108)$$

Der zweite Summand hat ein hohes Dämpfungsdekrement und einen sehr kleinen Vorfaktor. Er kann daher vernachlässigt werden. x_2 beschreibt die Bewegung des Rads. Beide Funktionen sind in Abb. 3 dargestellt.

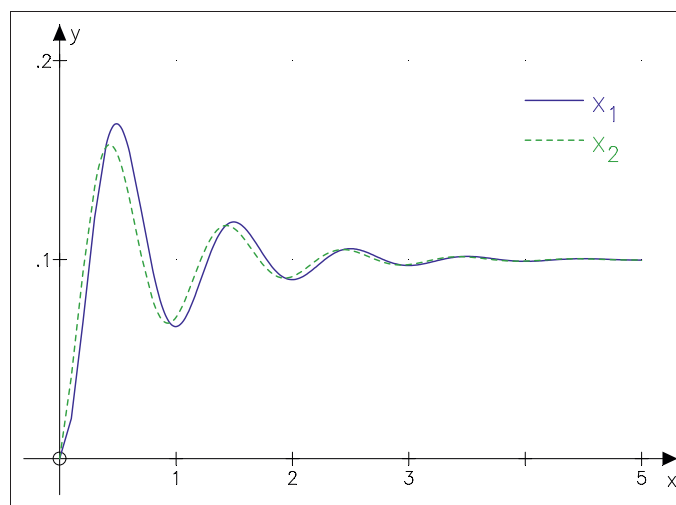


Abb. 3: Vertikale Lage der Karosserie (x_1) und des Rads (x_2)

Literatur

- [1] D. Kilsch. Ingenieurmathematik. TH Bingen, SoSe 2016.
- [2] D. Kilsch. Neuronale Netze. TH Bingen, SoSe 2016.
- [3] E. H. Moore. "On the reciprocal of the general algebraic matrix". In: *Bull. Amer. Math. Soc.* 26 (1920), S. 394–395.
- [4] R. Penrose. "A generalized inverse for matrices". In: *Proc. of the Cambridge Phil. Soc.* 51 (1955), S. 406–413.



Programmier-Aufgabe



Eine Programmier-Aufgabe

Von Ralf Herminghaus

Für diese Aufgabe betrachten wir einen Satz von drei Würfeln, nennen wir sie ROT, GRÜN und BLAU.

Die Würfel tragen die Augenzahlen:

ROT = (18 17 10 5 4 3)/Augensumme = 57

GRÜN = (16 15 14 9 2 1)/Augensumme = 57

BLAU = (13 12 11 8 7 6)/Augensumme = 57

Alle Zahlen von 1 bis 18 sind vertreten und die Augensumme jedes Würfels beträgt 57. Wenn man nun jeweils mit zweien der Würfel gegeneinander würfelt, welcher von ihnen hat wohl die größten Chancen, zu gewinnen?

Das kann man mit Wahrscheinlichkeitsrechnung leicht nachrechnen. Spielt man etwa die Würfel ROT gegen GRÜN, dann hat in $\frac{21}{36}$ der Fälle ROT die höhere Augenzahl.

Hier habe ich einmal alle möglichen Ergebnisse eines Wurfes ROT gegen GRÜN aufgeschrieben:

16	15	14	9	2	1
1	1	1	1	1	1
1	1	1	1	1	1
0	0	0	1	1	1
0	0	0	0	1	1
0	0	0	0	1	1
0	0	0	0	1	1

Wählt man den Würfel GRÜN gegen BLAU, so kommt man zu dem gleichen Ergebnis:

Im Spiel GRÜN gegen BLAU gewinnt GRÜN in 21/36 der Fälle.

13	12	11	8	7	6
1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
0	0	0	1	1	1
0	0	0	0	0	0
0	0	0	0	0	0

Wie steht es nun aber mit dem Spiel BLAU gegen ROT?

18	17	10	5	4	3
0	0	1	1	1	1
0	0	1	1	1	1
0	0	1	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1

Intuitiv würde man erwarten, dass BLAU gegen ROT noch deutlicher verliert, als gegen GRÜN.

Das Ergebnis überrascht daher:

BLAU gewinnt gegen ROT in 21/36 aller Fälle.

Jetzt kommt die Computer-Programmierung ins Spiel.

Ob es noch andere Augen-Belegungen der drei Würfel, die zu ähnlichen - vielleicht sogar noch deutlicheren Ergebnissen führen? Das kann man durch eine erschöpfende Suche tatsächlich einfach beantworten.

Ich habe dazu ein ziemlich unelegantes Programm aus elf ineinanderschachtelten FOR-Schleifen geschrieben.

Un-elegant - aber es hat immerhin funktioniert.

Es gibt genau acht Augen-Belegungen der Würfel, die die gleichen Eigenschaften haben, wie die eben untersuchte.

```

R: 18 17 10 5 4 3      57
G: 16 15 14 9 2 1      57
B: 13 12 11 8 7 6      57
-----
R: 18 17 8 7 6 1      57
G: 16 15 14 5 4 3      57
B: 13 12 11 10 9 2     57
-----
R: 18 13 12 11 2 1     57
G: 17 10 9 8 7 6       57
B: 16 15 14 5 4 3      57
-----
R: 18 13 8 7 6 5      57
G: 17 16 15 4 3 2      57
B: 14 12 11 10 9 1     57
-----
R: 18 12 9 7 6 5      57
G: 17 16 15 4 3 2      57
B: 14 13 11 10 8 1     57
-----
R: 18 11 10 7 6 5      57
G: 17 16 15 4 3 2      57
B: 14 13 12 9 8 1      57
-----
R: 18 11 9 8 6 5      57
G: 17 16 15 4 3 2      57
B: 14 13 12 10 7 1     57
-----
R: 18 10 9 8 7 5      57
G: 17 16 15 4 3 2      57
B: 14 13 12 11 6 1     57
-----

```

Als Mathematiker stellt man sich sofort die Frage nach Verallgemeinerungen dieser Fragestellung. Z.B. könnte man anstelle der würfel auch andere platonische Körper verwenden - etwa das Tetraeder?



In diesem Fall bekommt man nur sechs Lösungen mit einer maximal großen Asymmetrie in den Ergebnissen:
Hier hat die Asymmetrie die Größe 9/16.

```
-----  
R:  9  8  7  2   26  
G: 11  6  5  4   26  
B: 12 10  3  1   26  
-----  
R: 12  9  3  2   26  
G: 11  8  6  1   26  
B: 10  7  5  4   26  
-----  
R: 10  8  7  1   26  
G: 11  6  5  4   26  
B: 12  9  3  2   26  
-----  
R: 12  7  5  2   26  
G: 11 10  4  1   26  
B:  9  8  6  3   26  
-----  
R: 12  6  5  3   26  
G: 11 10  4  1   26  
B:  9  8  7  2   26  
-----  
R: 12  6  5  3   26  
G: 11  9  4  2   26  
B: 10  8  7  1   26  
-----
```

Der nächst größere Platonische Körper ist das Oktaeder - und da fängt es an, interessant zu werden. Nicht nur, dass die Rechenzeit für die erschöpfende Suche enorm ansteigt; die Anzahl der geschachtelten FOR-Schleifen wird auch allmählich unübersichtlich und macht das ganze Programm noch uneleganter, als es sowieso schon ist.

Die Suche liefert überraschenderweise auch nur ein einziges Ergebnis - und zwar mit einer maximalen Asymmetrie von 39/64.

```
-----  
R: 24 23 22  9  7  6  5  4   100  
G: 21 20 19 18 16  3  2  1   100  
B: 17 15 14 13 12 11 10  8   100  
-----
```


Leider sind damit die Grenzen meines kleinen Laptops erreicht. Und es sind noch immer zwei platonische Körper übrig, die man untersuchen könnte.

Wie steht es mit Dodekaeder und Ikosaeder?

Kann man die Suche nach Lösungen vielleicht intelligenter betreiben als ich es getan habe?

Liefert APL vielleicht die Möglichkeit, das Such-Programm eleganter zu formulieren als mit endlosen Stapeln ineinander geschachtelter FOR-Schleifen?

Hat irgendein genialer Mensch das Problem vor 200 Jahren schon mal mit Papier und Bleistift gelöst und ich weiß nur nichts davon?

Oder braucht es einfach nur einen schnelleren Rechner?

Auf die Antworten bin ich wirklich gespannt.



APL-Journal

37. Jg. 2018, ISSN 1438-4531

Herausgeber: Prof. Dr. Dieter Kilsch, APL-Germany e.V., Düsseldorf, Homepage: <https://apl-germany.de/>, E-Mail: d.kilsch@th-bingen.de

Redaktion: Dipl.-Volksw. Martin Barghoorn (verantw.), Lückhoffstr. 8, 14129 Berlin, E-Mail: barghoorn@zedat.fu-berlin.de

Verlag: RHOMBOS-VERLAG, Berlin, Kurfürstenstr. 15/16, D-10785 Berlin, Tel. (030) 261 9461, eMail: verlag@rhombos.de, Internet: <https://rhombos.de/>

Erscheinungsweise: halbjährlich

Erscheinungsort: Berlin

Druck: dbusiness.de GmbH, Berlin

Copyright: APL Germany e.V. (für alle Beiträge, die als Erstveröffentlichung erscheinen)

Fotonachweis: Martin Barghoorn (Umschlagseite 1 und 4, Seite 38, 51, 56, 61)

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutzgesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen. Eine Haftung für die Richtigkeit der veröffentlichten Informationen kann trotz sorgfältiger Prüfung von Herausgeber und Verlag nicht übernommen werden. Mit Namen gekennzeichnete Artikel geben nicht unbedingt die Meinung des Herausgebers oder der Redaktion wieder. Für unverlangte Einsendungen wird keine Haftung übernommen. Nachdruck ist nur mit Zustimmung des Herausgebers sowie mit Quellenangabe und Einsendung eines Beleges gestattet. Überarbeitungen eingesandter Manuskripte liegen im Ermessen der Redaktion.



Allgemeine Informationen

(Stand 2018)

APL-Germany e.V. ist ein gemeinnütziger Verein mit Sitz in Düsseldorf. Sein Zweck ist es, die Programmiersprache APL, sowie die Verbreitung des Verständnisses der Mensch-Maschine Kommunikation zu fördern. Für Interessenten, die zum Gedankenaustausch den Kontakt zu anderen APL-Benutzern suchen, sowie für solche, die sich aktiv an der Weiterverbreitung der Sprache

APL beteiligen wollen, bietet APL-Germany den adäquaten organisatorischen Rahmen.

Auf Antrag, über den der Vorstand entscheidet, kann jede natürliche oder juristische Person Mitglied werden. Organe des Vereins sind die mindestens einmal jährlich stattfindende Mitgliederversammlung sowie der jeweils auf zwei Jahre gewählte Vorstand.

1. Vorstandsvorsitzender

Prof. Dr. Dieter Kilsch,
Dumontstraße 12, 55313 Mainz,
Tel. 06131 6982200, E-Mail:
d.kilsch@th-bingen.de.

Beitragsätze

Ordentliche Mitglieder:

Natürliche Personen 32,- EUR*
Studenten / Schüler 11,- EUR*

Außerordentliche Mitglieder:

Jurist./natürl. Pers. 500,- EUR*

* Jahresbeitrag

2. Vorstandsvorsitzender:

Martin Barghoorn
Lückhoffstr. 8, 14129 Berlin,
E-Mail: barghoorn@zedat.fu-berlin.de

Schatzmeister

Jürgen Beckmann
Im Freudenheimer Grün 10
68259 Mannheim
Tel. 0621 7 98 08 40,
eMail: JBecki@onlinehome.de

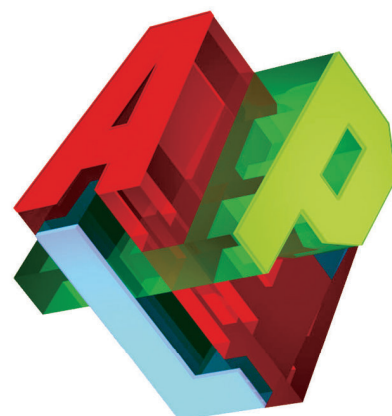
Bankverbindung

BVB Volksbank eG Bad Vilbel
BLZ 518 613 25
Konto-Nr. 523 2694

Hinweis:

Wir bitten alle Mitglieder, uns Adressänderungen und neue Bankverbindungen immer sofort mitzuteilen. Geben Sie bei Überweisungen den Namen und/oder die Mitgliedsnummer an.

<https://apl-germany.de/>





Wandbord

11,00

Dose ' Bienenwachs', 1 Stück

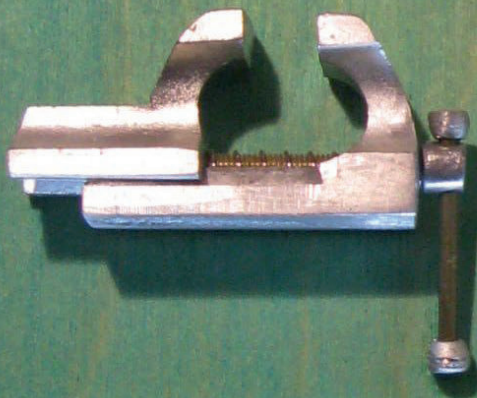
3,60

Dose ' Holzlasur', 1 Stück

3,60

Dose ' Holzkitt', 1 Stück

3,60



Schraubstock, funktionstüchtig

12,00 €



Amboss (Zinn)

8,50

Amboss (Zinn) mit Hammer & Zange

19,00



www.apl-germany.de
 APL Germany e.U.

