A DATA MODEL AND A TSCORE REPRESENTATION FOR THE TABSTAFF+ NOTATIONS FOR ABLETON PUSH

Markus Lepper
semantics gGmbH, Berlin, DE
post@markuslepper.eu

Baltasar Trancón y Widemann
semantics gGmbH, Berlin, DE
Technische Hochschule Brandenburg a.d. Havel, DE

ABSTRACT

The proposal "TABstaff+" by Wilde and White (2024) defines a collection of formats to notate the play on a grid-based tangible user interface. We propose a mathematical formulation for the underlying data model, and present an application to construct and render the corresponding models, using the frameworks for time-related modelling "tscore" and for meter-ruled rhythm notation "metricSplit".

1. CONTEXT

The proposal "TABstaff+" by Wilde and White (2024) defines a collection of formats to notate the play on a grid-based tangible user interface (TUI). [1] Their proposal comprises three formats, namely TAB+ = tablature notation, Staff+ = annotations for execution, to be added above a staff of conventional music notation, and Charts+ = diagrams in the style of guitar chord diagrams. The tablature notation TAB+ is inspired by a proposal by Chris Sperren. [2] (See our Figure 3 for an example of our version of Charts+ and Figure 4 for TAB+. The elements of Staff+ are just numeric indications of the rows and columns to press, added above a conventional note system, and not discussed here.)

Their article discusses the "educational implications" of these notations, and describes their development in "multiple iterations [of tests] on various grid-based instruments [by] music composition and production students [...] with varying musical abilities."

In that article, the syntax and semantics are defined informally, by prose language and graphic illustrations. As a preparatory step for implementing a digital tool, an analysis of the underlying data model is useful, especially when a common data model can be identified for all three proposed formats. Furthermore, mathematical remodelling always brings more clarity also into human discourse and inexorably exhibits blurry areas of concepts. This article discusses such a data model to capture TAB+ and Charts+.

The next section describes the proposal of Wilde and White in more detail; then this is translated into a mathematical data model. The subsequent sections describe the `tScore` input format, the tool implementation, and the graphic rendering algorithms, their theory and their practical caveats.

2. ABLETON PUSH AND TABSTAFF+

In their experiments, Wilde and White apply the proposed notations to the "Ableton Push" instrument, to other grid-based TUIs with a similar design, and to grid-based graphic user interfaces (GUIs) on touchscreens.

The Ableton Push is the origin of the described TUI and a full-fledged digital synthesizer and sample player, see Figure 1. With the same TUI input, much cheaper hardware exists restricted to the MIDI controller function. We use "Novation Launchpad MK3" to run experiments with our implementation.

The main interface realized by the Ableton Push is a matrix of eight times eight push buttons. Following the original manuals [3], the buttons are called *pads* and the matrix is called *grid* in the following. The coordinates of the pads are given as pair of numbers from 1 to 8. The order is "row before column" = "y before x", i.e. the way to address a mathematical matrix, not the Cartesian coordinates. The rows count bottom up, and the columns from left to right.

The TUI can be operated in different modes, where the pads have different meanings. These modes are called *layout* by the original hardware manual and by Wilde and White. For instance, emitting MIDI note-on and note-off events can be assigned to a pad, but also triggering the start of a prepared sound clip. The proposed notations TAB+ and Charts+ and the corresponding `tScore` input format give the physical coordinates of the pressed pads and thus are *completely independent* of the roles assigned to them. Format Staff+ is different, because it is added to conventional staves in Common Western Notation (CWN). It is not treated in this article.

Additionally the TUI hardware can have a display area and further buttons. These can be used to modify the layout on the fly while playing. In the `tScore` format these modifications are represented by events in a dedicated configuration voice named "`conf`", see below in section 4.

3. MATHEMATICAL RE-MODELLING

Table 1 shows a possible mathematical re-modelling of the information contents of TAB+ and Charts+. (As a mere analysis of the information contents of the proposals published in [1] it is independent from its `tScore` implementation as described in the next section, but it already follows the same principles.)

The first block of lines gives the data types for the events.

In `tScore` every event is identical to one particular combination of voice and timepoint. (All further event param-

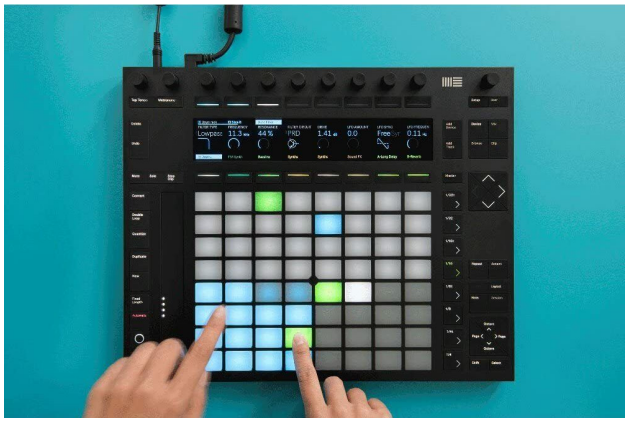


Figure 1. The "Ableton Push" synthesizher with its main input device, the pads grid. (Source: ableton.com)

$$
\begin{array}{l}
\mathsf{Hand} = \{\mathtt{md}, \mathtt{ms}\} \\
\mathsf{FNumber} = \{1..5\} \\
\mathsf{Row} = \{1..8\} \\
\mathsf{Column} = \{1..8\} \\
\left|\begin{array}{ll}
\mathbf{V} & \textit{// given set of voices} \\
\mathbf{T} = \mathbb{N} & \textit{// given set of timepoints}
\end{array}\right. \\
\mathbf{E} = \mathbf{V} \times \mathbf{T} \quad \textit{// derived set of events} \\
\textit{// Parsed content of } \mathtt{tScore} \textit{ events:} \\
\mathsf{pad} : \mathbf{E} \nrightarrow \mathsf{OPT}(\mathsf{Row} \times \mathsf{Column}) \\
\qquad \textit{// = } \perp \textit{ stands for pause} \\
\mathsf{hand} : \mathbf{E} \nrightarrow \mathsf{OPT\ Hand} \\
\qquad \textit{// = } \perp \textit{ stands for "hand ad lib explicitly"} \\
\mathsf{finger} : \mathbf{E} \nrightarrow \mathsf{FNumber} \\
\mathsf{finger_X} : \mathbf{E} \nrightarrow \mathsf{OPT\ FNumber} \\
\mathsf{finger_X} = \big((\mathsf{dom\ pad} \cup \mathsf{dom\ hand}) \times \{\perp\}\big) \oplus \mathsf{finger} \\
\qquad \textit{// = } \perp \textit{ stands for "not longer valid"} \\
\\
\textit{// Totalize any function } F : \mathbf{E} \nrightarrow \mathsf{OPT}\ X\textit{:} \\
\overline{F} = \{v, t \bullet (v, t) \mapsto \mathbf{if}\ t = 0\ \mathbf{then} \perp \mathbf{else}\ \overline{F}(v, \mathsf{pred}\, t)\} \\
\qquad \oplus\ F \\
\\
\textit{// Indispensable invariants:} \\
\mathsf{dom\ finger} \subseteq \mathsf{dom}(\overline{\mathsf{hand}} \rhd \{\perp\}) \\
\mathsf{dom\ finger} \subseteq \mathsf{dom}(\overline{\mathsf{pad}} \rhd \{\perp\})
\end{array}
$$

Table 1. Syntactic and semantic data

eters are only adjoined and not constitutive.) On the time points an order is defined, and thus the notions of minimum and maximum.

The input data are partial functions from events into different domains. Here they specify the pad to press and optionally the hand and the finger to use. The range of these mappings is an optional data type: Taking the variant "no data" = $\perp$ has different meanings: Used instead of a pad it means to play a *pause*. With hand and finger it means that this is not (or no longer) prescribed.

For every event function F, the smallest fixpoint of the function definition $\overline{F}$ delivers the value specified by F at the given timepoint, or the value specified most recently before, or $\perp$. In a polyphonic context, $\overline{F}$ is especially necessary to analyze overlapping asynchronous events.

$$
\begin{array}{l}
\textit{Coord} ::= \text{"}\mathtt{1}\text{"} \mid \ldots \mid \text{"}\mathtt{8}\text{"} \\
\textit{Hand} ::= \text{"}\mathtt{r}\text{"} \mid \text{"}\mathtt{R}\text{"} \mid \text{"}\mathtt{D}\text{"} \mid \text{"}\mathtt{d}\text{"} \\
\qquad \mid \text{"}\mathtt{S}\text{"} \mid \text{"}\mathtt{s}\text{"} \mid \text{"}\mathtt{L}\text{"} \mid \text{"}\mathtt{l}\text{"} \mid \text{"}\mathtt{X}\text{"} \\
\textit{Finger} ::= \text{"}\mathtt{1}\text{"} \mid \ldots \mid \text{"}\mathtt{5}\text{"} \\
\textit{Pause} ::= \text{"}\mathtt{\%}\text{"}\ \textit{Hand}^? \\
\textit{Keypress} ::= \textit{Coord}\ \textit{Coord}\ \textit{Hand}^?\ \textit{Finger}^? \\
\textit{FChange} ::= \textit{Hand}\ \textit{Finger}^? \mid \text{">"}\ \textit{Hand}^?\ \textit{Finger}^? \\
\textit{Event} ::= \textit{Pause} \mid \text{"}-\text{"} \mid \textit{Keypress} \mid \textit{FChange}
\end{array}
$$

Table 2. Grammar rules for `tScore` events

The finger numbers are not "sticky" but only valid for the one notated event. This must be modelled explicitly by deriving $\mathsf{finger_X}$.[1]

The last two lines give invariants, the violation of which leads to a rejection by the implementation: fingers may not be specified without a hand, and fingers may not be specified for pauses. (See appendix A below, which discusses some parallels in conventional CWN piano notation.)

4. TSCORE INPUT FORMAT

`TScore` is a concept and an implementation for notating arbitrary data models which are organized along a time axis. It can be written conveniently and fast, and is equally well readable by computers and humans. [4] [5]

As usual in computer engineering, it is called a *meta-meta-model*. It allows easy definition of *meta-models*, which define the input grammar, the data structures, semantical invariants, and processing paths for a collection of encoded pieces, the *models* finally. (In this sense the formulas in Table 1 are a meta-model for all TABstaff+ data.) For a typical application to graphic avant-garde notation see [6]; for a collection of very different examples see [7].

The `tScore` meta-model which covers the TABstaff+ data model, our versions of rendering to tablature and charts, and the Java implementation, are called `squarePads`. Table 2 shows the grammar of the events in the `tScore` input, and Figure 2 shows a simple example.

The voice named "`conf`" determines the grid layout, and thus assigns roles to the pads of the grid. With the very first event of this voice some attributes are necessary. Later events may carry subsets of these parameters, to change the layout dynamically during the performance, see section 8.3.

The voice named "`nota`" plays different roles for different output media: It can determine the staves and clefs shown in tablature notation, and the size of the generated charts.

All other voices (with arbitrary but distinct names) hold the information what to play: Every event is either a pause "`%`", or a prolongation "–" of the preceding event, or a pair of row and column number, meaning a pad press. (Normally a pad is meant to be held down until the next event, but of course this may be superfluous for contexts where

[1] The name of the domain "FNumber" has been chosen to stress that it does not stand for a real-world "finger". For this you need a pair Hand × FNumber.

```
PARS sola
lingua=en
formaSuprema=eu.bandm.music.applications.squarepads.SquarePads
matrixClavium.tabulatura.vocesTangentes = faciliusLectu
// = Shift adjacent noteheads from different voices apart for easier reading.
matrixClavium.tabulatura.differe = (-1, +1)
// = On conflicts shift upper voice to the left and lower to the right.

T           0                                   1
VOX conf    horizontal+&fixed-&inKey-&thirds&minor&c2
VOX nota    ABαβ
VOX v1      11      21L3    21L3       >4      %
VOX v2      51R     51R1    >R2        L       %

          // Means
          // First row first column, hand and finger ad lib.
          // Fifth row first column, right hand, finger ad lib.
                  // Second row first column, third finger of left hand.
                  // Fifth row first column, first finger of right hand.
                          // Re-attack same button with same finger.
                          // Silently change to second finger of right hand.
                                    // Silently change to finger 4,  same hand.
                                    // Silently change to left hand, any finger.
```

Figure 2. A small `squarePads` `tScore`. The corresponding chart rendering is in Figure 3.

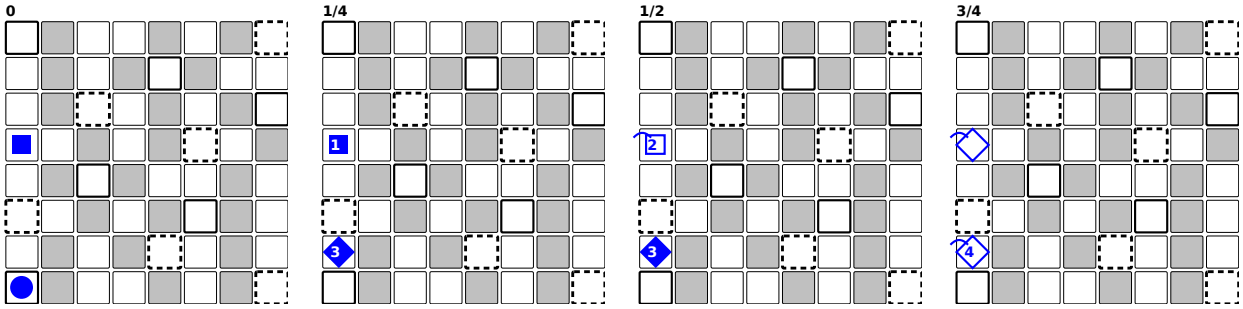


Figure 3. Rendered chord charts: hand is ○ unspecified □ right ◇ left

only the onset of the pressing [= the MIDI note-on event] is evaluated as a trigger.)

The pause symbol and the pad coordinates may be followed by a letter indicating the playing hand.

A pad coordinate can be followed by a finger number, from "`1`" to "`5`", as with piano notation. The hand information is treated sticky: it stays valid until explicitly ended by another indication, which may be "`X`", meaning "hand unspecified".

A finger number may only be added when a hand is currently specified, by this or an earlier pad-press event.

The event "`>`" indicates a silent finger change. It may be followed by a hand indication or a finger number or both. The preceding event must not be a pause or its prolongation.

The `tScore` input format is basically *monophonic*: every combination of timepoint and voice can carry at most one single pad information. Input formats which allow *chords* of pressed' pad are included in the implementation as an alternative, but this is only "syntactic sugar" and is resolved to automatically supplied additional voices.

5. CRITICAL PROPERTIES

One of the most important roles of a mathematical model as shown in Table 1 is as a basis for precise definitions of model properties and thus categories. Usually an application context or a historic or personal style of writing is naturally characterized by a restriction to a particular subset of models. Such a characterization can be discovered by algorithmic filtering by the defined properties.[2] Such properties are useful not only for notated music, but also for interactive performances, as long as they are translated into the data model.

Important for new kinds of manual input devices, but also for particular styles of writing for conventional keyboard, are PressPattern and DIGITINONFRACTI: they define the geometric arrangements of multiple pads which must be pressed by the same finger, and whether a composition or performance sticks to it. pressPattern_easy is an example where one single pad or two or four all-adjacent pads are pressed.

6. CHART NOTATION

The current demo implementation generates graphic representations which resemble the chord charts as widely used for guitars etc.,as proposed by Wilde and White. The di-

[2] The names of all properties are chosen from the Latin language to fit nicely into the system proposed as LMN [8]. Read as an LMN module, the complete name of all these properties is prefixed by MATRIXCLAVIUM.

$t : \mathbf{T}, r : \mathsf{Row}, c : \mathsf{Column}, h : \mathsf{Hand}, f : \mathsf{FNumber}$

$\textsc{clavisPerVoces}(t, r, c) = \{v \mid \overline{\mathsf{pad}}(v, t) = (r, c)\}$
// = set of voices targeting the given pad simultaneously
$\textsc{perNotasMultas} = \{t, r, c \mid \#\textsc{clavisPerVoces} > 1\}$
// = pads pressed by multiple voices simultaneously
$\textsc{clavesPerNotamSimplicem} \iff \textsc{perNotasMultas} = \varnothing$
// = every pad appears at every timepoint not more than once

$\textsc{clavisPerDigitos}(t, r, c) = \{v \mid \overline{\mathsf{pad}}(v, t) = (r, c) \bullet (\overline{\mathsf{hand}}(v, t), \overline{\mathsf{finger}_{\mathrm{X}}}(v, t))\} \setminus (\mathsf{Hand} \times \{\bot\})$
// = set of fingers which press a pad simultaneously
$\textsc{clavisPerDigitosMultos} = \{t, r, c \mid \#\textsc{clavisPerDigitos}(t, r, c) > 1\}$
// = set of pads which pressed by multiple fingers simultaneously
$\textsc{clavesSimplices} \iff \textsc{clavisPerDigitosMultos} = \varnothing$
// = no pad is ever pressed by multiple fingers simultaneously

$\textsc{digitusInClaves}(t, h, f) = \{v \mid \overline{\mathsf{hand}}(v, t) = h \wedge \overline{\mathsf{finger}_{\mathrm{X}}}(v, t) = f \bullet \overline{\mathsf{pad}}(v, t)\}$
// = set of all keys pressed by the given finger
$\textsc{digitusInMultosClaves} = \{t, h, f \mid \#\textsc{digitusInClaves}(t, h, f) > 1\}$
// = set of fingers which press multiple pads simultaneously
$\textsc{digitiSimplices} \iff \textsc{digitusInMultosClaves} = \varnothing$
// = no finger ever presses multiple pads simultaneously

$\textsc{manusCertae} = \bot \notin \mathsf{dom}\ \overline{\mathsf{hand}}$
// = for every pad press and every pause the hand is (sticky) prescribed
$\textsc{pausaCumManu} = (\mathsf{pad}^{\sim}(\!|\{\bot\}|\!)) \triangleleft \mathsf{hand} \neq \varnothing$
// = at least one pause carries an explict hand prescription.
$\textsc{digitiCerti} = \mathsf{dom}\ \mathsf{pad} \subseteq \mathsf{dom}\ \mathsf{finger}$
// = for every pad press the finger is prescribed
$\textsc{sineDigitis} = (\mathsf{finger} = \varnothing)$
// = for no single pad press the finger is prescribed
$\textsc{digitiCerti} \implies \textsc{manusCertae}$

// Indispensible invariants by physiology:
$\mathsf{PressPattern} = \mathbb{PP}(\mathbb{N} \times \mathbb{N})$
$\mathsf{pressPattern\_easy} = \{\{(0,0)\}, \{(0,0),(0,1)\}, \{(0,0),(1,0)\}, \{(0,0),(0,1),(1,0),(1,1)\}\}$
// = set of patterns of pads which can be pressed simultaneously by one finger.
$instantiatePattern : \mathsf{PressPattern} \to \mathbb{PP}(\mathsf{Row} \times \mathsf{Column})$
$instantiatePattern(P) = \bigcup p \in P \bullet \{a, b : \mathbb{N} \bullet \{(c, d) \in p \bullet (a + c, b + d)\} \cap \mathbb{P}(\mathsf{Row} \times \mathsf{Column})\}$

$\textsc{digitiNonFracti}(P : \mathsf{PressPattern}) = \forall t, h, f \bullet \textsc{digitusInClaves}(t, h, f) \in instantiatePattern(P)$
// = indicates that the whole score requires to play not more than the given patterns.
$\textsc{digitiNonFracti}(\{\{(0,0)\}\}) \iff \textsc{digitiSimplices}$

Table 3. Characterizing properties w.r.t. fingers and hands.

atonic pitches, the chromatic steps between, and the root class pitches are represented by combinations of the background color and outline thickness of the pad symbols. This is in analogy to the state of the color LEDs in the hardware pads.

The pads pressed by the left and right hand are marked by rectangle symbols, upright for right hand, diamond-shaped for the left hand. Circles stand for notes without hand indication. Finger numbers are centered, silent finger changes are symbols with outlines only and no fill.

The current implementation generates single SVG files, see Figure 3, and one animated SVG file, assuming BPM 60.

7. TABLATURE NOTATION

The principle of tablature notation has been well-known for more than five centuries, in very different variants. [9] It is a notation of the *physical operation* on fretted string instruments: The lines of the staff represent the strings; the horizontal position from left to right represents the flow of time; the numbers put into the staff represent the fret to press, plus possibly additional symbols like finger indication and *laissez vibrer*.

Much later, this format has been transferred to key instruments (Klavarskribo [10], piano roll [11], etc.)

Transferring this format to a two-dimensional TUI seems sensible, but brings two major challenges:

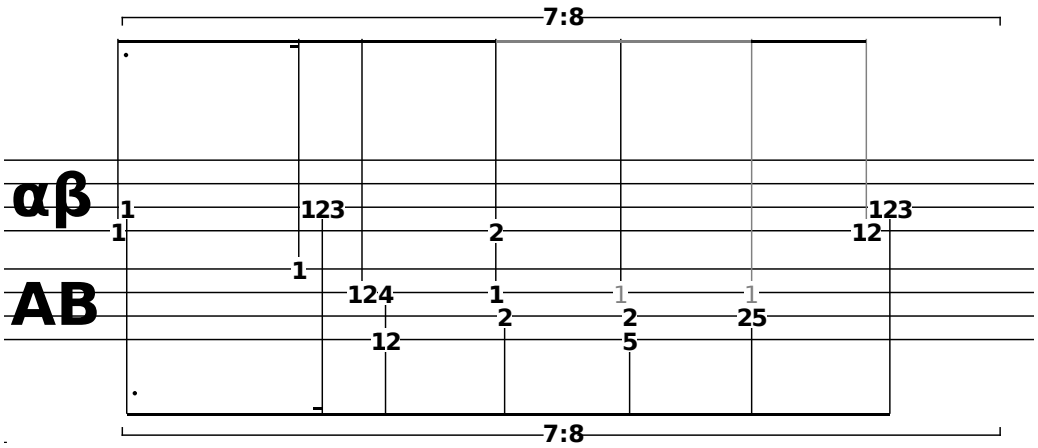


Figure 4. Collision Corrections in the Rendered Tablature Notation

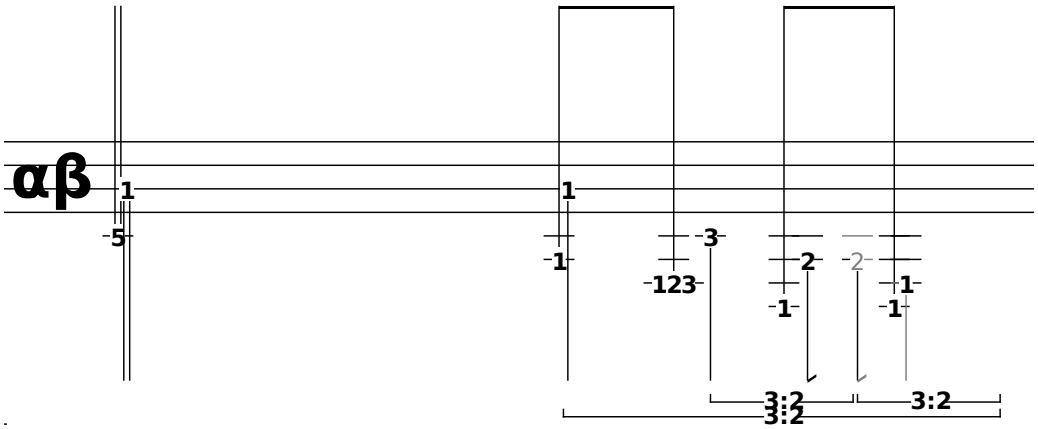


Figure 5. Ledger Lines in the Rendered Tablature Notation

(A) The staff lines stand for the rows of the grid, the numbers therein for the columns to press. In contrast to frets, *multiple* numbers on the same combination of lineola and timepoint appear frequently.

(B) In particular contexts, we may wish to notate both hands in the same staff, with upward vs. downward stems.

Both these new properties cause the need for *collision correction* when programming automated graphical layout.

Furthermore, (C) Wilde and White propose *ledger lines* for spurious outbreaking events, when normally only the lower or the upper half of the grid is used and represented by four lines. (The staff lines in historic tablature notation systems represent *all* stopped strings of the instrument—we have not found any example which uses ledger lines.)

7.1 Graphic Positioning Problems in Detail

In `squarePads`, we have implemented the following strategy for the graphic appearance: The relation of the column numbers to one of the two voices (right hand vs. left hand = up stem vs. down stem) is always unambiguously clear because the sequence of numbers is printed centered on the stem, see the sequence "124" in Figure 4. (An alternative is to place the numbers to the left or right, like noteheads in conventional staves, but this turned out much less readable. See [2] for examples.)

Tying noteheads as in CWN can create hardly solvable collisions among the arcs and between arcs and staff lines. Instead, in our rendering all non-leading tied noteheads (which do not represent a new attack but merely a longer holding of the pad) are printed in gray instead of black.

In case of collisions, the two stems must be moved apart by a certain distance. This distance is calculated according to the formulas in Table 4. When noteheads overlap, the shift is unavoidable, see the example in Figure 4. Figure 5 shows an example with ledger lines and half notes indicated by double stems.

This is different when the two columns of numbers only touch, without overlapping. In Figure 6, the two voices in example (a) are unambiguous and printed without any shift.

Contrarily, columns (b) and (c) would be ambiguous if printed without shift, because they contain multiple neighboring pairs of rows each.

Examples (d) and (e) are not ambiguous, but probably hard to read prima vista. So they get a shift if the configuration parameter TABULATURA.VOCESTANGENTES takes the value FACILIUSLECTU.

The notation property TABULATURA.DIFFERRE indicates for both voices in which horizontal direction their stem will contribute to a necessary shift.

$$
\begin{array}{l}
nhUp, nhDown : \mathsf{Row} \to \mathbb{P}\mathsf{Column} \\
U = \mathsf{dom}\ nhUp \rhd \varnothing \\
D = \mathsf{dom}\ nhDown \rhd \varnothing \\
A = U \cup D \\
i = \mathsf{min}\ U \\
a = \mathsf{max}\ D \\
corr = \textbf{if}\ i > a + 1\ \textbf{then}\ 0 \\
\qquad \textbf{else if}\ i = a + 1\ \textbf{then}\ touching \\
\qquad \textbf{else}\ \mathsf{max}\ \{r \in i..a \bullet \#nhUp(r) + \#nhDown(r)\}
\end{array}
$$

$$
\begin{array}{l}
\text{MATRIXCLAVIUM.TABULATURA.VOCESTANGENTES} : \{\text{FACILIUSLECTU}, \text{NECESSARIA}\} \\
touching = \textbf{if}\ \text{VOCESTANGENTES} = \text{FACILIUSLECTU}\ \textbf{then}\ [\ \#A > 2\ ] \\
\qquad \textbf{else}\ [\ \#A \cap succ(\!|A|\!) > 1\ ]
\end{array}
$$

$$
\begin{array}{l}
Shift = \{-1, 0, +1\} \\
\text{DIFFERRE} : (Shift \times Shift) \setminus \mathbf{ID}_{Shift} \\
x_{\text{up}} \quad = x + corr * \pi_1(\text{DIFFERRE}) * ([\pi_2(\text{DIFFERRE}) = 0] + 1) * WIDTH\_OF\_DIGITS/4 \\
x_{\text{down}} \quad = x + corr * \pi_2(\text{DIFFERRE}) * ([\pi_1(\text{DIFFERRE}) = 0] + 1) * WIDTH\_OF\_DIGITS/4
\end{array}
$$

Table 4. Tablature collision correction, at one particular time point with nominal horizontal position x.

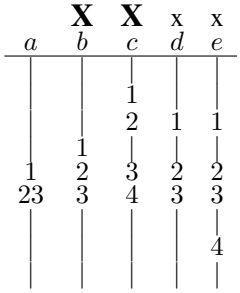


Figure 6. Real ambiguities (**X**) vs. presumed reading problems (x), caused by adjacent noteheads

Figure 7. Proposed noteheads to indicate columns graphically: (A) one event to press all pads in a row of eight columns. The design of the arrow heads requires one additional gap only in two cases: "4 (5) appears without direct neighbors but with successors (predecessors)". See (B) which addresses columns 2, 4, 6, and 8.
(C) and (D) show an event which presses all keys in a row in a four-column grid, in two different notation proposals.

Figure 4 shows the different collision cases. (In the current implementation, only collisions at the *same timepoint* are corrected. All timepoints are arranged strictly proportionally, and calculating the overall width is not yet automated in `tScore` anyhow.) [3]

Generating the flags, beams, proportion brackets, and prolongation dots and ties is done by metricSplit, our algorithm which can render any sequence of rational duration values to CWN. [12, 13]

The attempt to add fingering information by further decimal digits would imply hard-to-read overloading. (Numbers 1–5 are usual in keyboard notation, letters p, i, m, a in right-hand guitar.) An alternative is to use graphic symbols as noteheads for column information, for instance as in Figure 7. But practical experience is still missing.

8. PITCH LAYOUT AND ITS DYNAMIC CHANGE

We mentioned above, that the pad press events in TAB+, Staff+, and `tScore` are notated by their physical coordinates and thus completely independent from the meanings of the pads, and that the relation from all 64 grid pads to their meanings is called *layout* by the original hardware manual. Some of these layouts are called *64 notes layout* [3, section 7]: All $8 * 8$ pads generate MIDI note-on and note-off events; each pad is assigned to one particular MIDI key number; its lighting indicates its role in the currently intended diatonic scale and appears on the LEDs of the physical device and in the graphic boxes produced by Charts+. The possible configurations will be called *pitch layouts* in the following.

A challenge for re-modelling is the specification of the *dynamic changes*: While a session is running, the player can press dedicated buttons to change the parameters of the pitch layout on the fly. This is realized by `tScore`

[3] The demo application is currently just a proof of concept: Only one measure is rendered, which must extend from timepoint $0/1$ to $1/1$ and be in a $4/4$ meter. By resizing the window of the application, you can explore the visual appearance at different display widths.

by the dedicated voice "conf" which contains dedicated commands, see Figure 2 above.

While this pitch layout is independent from all notation systems described above, it must be implemented whenever a sound synthesis shall be addressed by some software. Furthermore, its remodelling (with mathematical means) by interpreting the original specification (in plain language only) is instructive and thus included in this article as a second, independent part.

8.1 The Original Specification

The Ableton Push 3 Manual [3] gives only an informal and very brief description how a pitch layout is changed dynamically:

> 7.1 Playing in Other Keys
> You can press the Scale button to switch to a different key and scale. You will see the available keys and scales in the display.
>
> Use the [..] buttons to select the key [and] to select a scale.
>
> The leftmost encoder changes the layout of the pad grid:
> 4ths – When moving up to the next pad vertically, each pad is a fourth higher.
> 3rds – When moving up to the next pad vertically, each pad is a third higher.
> Sequent – When moving to the right, each pad is in order based on the selected scale. [4]
>
> [..] When set to In Key, only the notes that are part of the selected key will be available on the pads. When set to Chromatic, all notes can be played; however, pads that contain notes that are not part of the key will be unlit.
>
> [..] When Fixed is on, the notes of the pad grid remain in the same position when you change keys, e.g. the bottom-left pad will always play C (unless the key does not contain a C, in which case the pad will play the nearest note to C). When Fixed is off, the notes on the pad grid shift so that the bottom-left pad always plays the selected root note.

Fundamental questions are left open:

(Q1) What are a "third" and a "fourth"? (This question is deeply related to the structures of the selectable "scales", which are not mentioned at all in the above-quoted text. The illustrating screen shot shows that a "Whole Tone scale" is offered. This does not contain any (pure) fourth!)

(Q2) What does "nearest note to C" mean?

(Q3) Switching the root pitch in non-fixed mode can bring any pitch x to the "bottom-left" pad $(1, 1)$. Then changing (while continuing the session) to fixed mode and then switching the root again will probably leave x on position $(1, 1)$, not "C"?

(Q4) Does the "Fixed" behavior of letting the maximal number of pitches in place also apply when changing the distance?

[4] This cannot be a definition of this mode because its always true in horizontal mode. The "push-2 manual" at https://www.ableton.com/en/manual/using-push-2/ says "The Sequent layout puts all notes sequentially in order." Indeed the right-most pad in every row is one pitch below the left-most pad in the row above, if any.

There are different possibilities to deal with such an incomplete specification. In the following we try to *explicate* it, by constructing different alternative interpretations and their algorithmic implementations. This allows to explore their limitations and consequences.

An alternative would be to do *reverse engineering* of the commercial product. But in contrast to a specification, its behavior cannot be expected to stay stable, and the discussion would be restricted to the single solution chosen by the engineers of that particular product and possibly miss the more fundamental rules and dependencies.

8.2 Static Pitch Layout by Parameters

When creating a pitch layout (= assignment of midi pitches and lighting states to pads) from scratch, it is defined completely by the parameters scale, root, distance (appearing as "4ths", "3rds", and "Sequent" in the manual text quoted above) , InKey (see "In Key" above), and orientation, as declared in Table 6. (All following text covers only the case orientation $=$ `horizontal`. The vertical layout can be derived easily by post-processing, namely by swapping rows and columns.)

(In the following the term *root* is reserved for the pitch class used as the root of the scale, and *base pitch* stands for the concrete pitch which is given as parameter and is assigned to the offset zero, as used technically in our model in Tables 6 and 7.)

All these parameters must be defined to create an initial (pitch) layout, and they define it completely. The algorithm has only to answer question (Q1) from the list above.

But these parameters can also be altered by the user dynamically, while playing. (We assume that only the change of one of these parameters occurs at a particular time point, but a second change can follow arbitrarily soon.)

In this case we call the currently active pitch layout the *preceding layout*. There is a meta-parameter Fixed of boolean type. When it is set to `false`, the new pitch layout is created from the current parameter values as if it were an initial one. But when Fixed $=$ `true`, then the new pitch layout is derived from the preceding layout, to minimize the number of pads affected.

Tables 6 and 7 start with a possible algorithm for the construction of an initial layout: The function $K_{\mathrm{R}}(s, k, S, c)$ delivers the MIDI offset (relative to the base pitch) pitch for the pad in column c, when the row starts with offset s, the InKey mode is k and S is the selected scale. Any scale is given as a pair of a set of indexes from 0 to 11 and a single such index. The set is similar but not totally equivalent to the well-known *pitch class set*. It indicates the steps of the chromatic scale which are contained in S. Step number 0 is contained by default. When the second component is set to a number $\neq 0$, it indicates a step of the scale which shall be highlighted as a *repercussa*, which is the somehow second-important step of the scale. This highlighting is our extension for a better orientation on the grid of pads.

For $k =$ `false` the chromatic pitches follow in dense order starting with s; for $k =$ `true` the list of all pitches of the scale $pitches(S)$ is constructed. Then this is restricted to begin with the given start point of the row, and indexed

by the column number c. (In Z in general, and as used in this article, sequences start with the index 1, see the appendix.)

The role of the pad (for their lighting) is derived from the MIDI pitch by the function $L_{\mathrm{R}}(s, k, S, c)$. Please note that none of the two functions needs to know the currently ruling value of the base pitch—they work with indices relative to the current base pitch. The MIDI key numbers effectively emitted with pad press events are calculated by adding the current value of base pitch to this index.

It is more problematic to find the first pitch for every row, realized by the vector V:

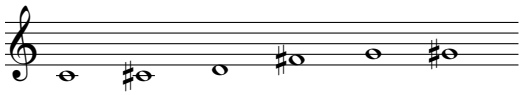
In every *initial* construction, the pad at $(1, 1)$ is assigned the selected base pitch—thus to the relative coordinate 0. When distance $=$ `sequent` then the next higher row will seamlessly continue the sequence of pitches and roles. In the other cases, a pitch "a third above" or "a fourth above" must be found. The function $N(s, k, S, d)$ delivers the start point for the next higher row of a row starting with s—again k is the InKey mode, S the scale, and d the selected distance.

N_1 to N_3 implement *variants*: N_1 delivers the next diatonic key with a distance equal or larger than a minor third or a fourth, respectively, above s.

N_2 is only applicable to InKey $=$ `false` and distance $=$ `fourths`, and simply goes five chromatic steps up. This is the only case not using `sequent` where a non-diatonic pitch can appear in the first column.

N_3 takes the historic origin of the interval names in a verbatim way and delivers the second or third successor of s, relative to S.

All variants can be appropriate to particular use cases. For the scale S starting with the pitch classes



it holds that

$$N_1(0, \_, S, \texttt{thirds}) = N_1(0, \_, S, \texttt{fourths}) \equiv \mathrm{F}\sharp$$
$$N_2(0, \texttt{false}, S, \texttt{fourths}) \equiv \mathrm{F}$$
$$N_3(0, \_, S, \texttt{thirds}) \equiv \mathrm{D}$$
$$N_3(0, \_, S, \texttt{fourths}) \equiv \mathrm{F}\sharp$$

The function initV creates the vector of type V of the starts of all rows by recursively calling N.

The current global state of the matrix is completely captured by an instance from G. Function initG creates such a state for the given parameters, from scratch, and $K(G, r, c)$ and $L(G, r, c)$ deliver the MIDI key number and the lighting for the pad in row r and column c.

8.3 Dynamic Change of Pitch Layout Parameters in Fixed mode

The dynamic changes of pitch layout parameters, i.e. during playing, when the device is in a particular pitch layout state, can be much more complicated. With the meta-parameter Fixed set to `false`, it has the same effect as creating a pitch layout from scratch, as described in the preceding section. But setting Fixed to `true` aims at a layout change with a minimal number of pads affected—to realize a "principle of minimal changes." This is ergonomically sensible: For instance, when changing the playing context only by shifting the root pitch class up by one or two positions in the circle of fifth, the player often wants to continue playing the common (and thus un-affected) pitches on the same pads as before. This corresponds to the handling of a traditional piano or organ keyboard, where only (in the mental reservoir maintained by the player) the white key f is replaced by the black key f-sharp, etc.

change…	…affects MIDI	…lighting
chromatic mode, $k =$ `false`		
base pitch	-	X
scale	-	X
distance	H	H
diatonic mode, $k =$ `true`		
root	X	-
scale	X	(x)
distance	H	H
diatonic $\longleftrightarrow$ chromatic	X	X

Table 5. Pitch layout parameters and the effects of changing them in Fixed mode. X = always changes. H = always changes, but not the lowest row. (x) = possibly, when lengths or repercussa of scales differ, - = no change.

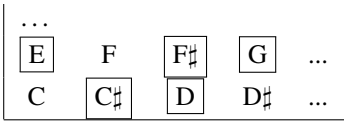
While the basic idea appears self-evident and ergonomic at first sight, its realization is complicated and indeed brings up inherent contradictions. Table 5 gives a survey of the minimal and possible effects. We affirm question (Q3) from above and thus include the vector $v : V$ in the global state G, to satisfy all change requests in the context of the preceding layout. The corresponding function change(g, x) in Table 7 is *overloaded*: The *type* of its second parameter x determines which component of the global state g shall be changed.

Taking the original text-only specification (as cited above) seriously, results in complicated behaviour. You can use the application (see section 9) for own experiments—no special hardware is required.

8.3.1 InKey $=$ `false`

In chromatic mode, changing scale and root is easy: All pads keep their pitch role, only the lighting changes, according to a shifted version of the corresponding initial layout. (In the formula, only the vector of the row starting points $v : V$ must be adjusted to compensate the change to the new pitch base q when emitting MIDI keys.)

We affirm question (Q4): Changing distance will change all pad assignments except in the lowest row. It is problematic due to question (Q1) from the list above, and because pad $(1, 1)$ is not always diatonic. The pitch layout



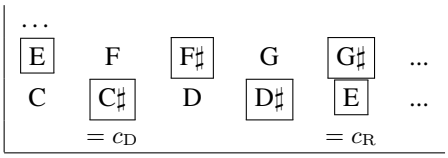
is reached when starting in C major and changing in fixed mode the root to D.

Our solution treats the root pad in the lowest row (its index is c_R in Table 7; $pitches((\varnothing, 0), x)$ is the simply the sequence of all multiples of 12 which are equal or greater than x.) in the same way as the initial construction treats pad $(1, 1)$, and then calculates back for the first columns.

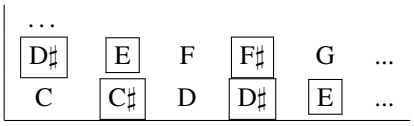
If no root pad appears in the bottom row, the first diatonic pad is taken as that anchor, its index is c_D.

Our first programming attempt used only this pad (by setting $c_1 = c_\mathrm{D}$), but this is not a good and stable solution for all cases, because (only) the distance `thirds` depends on the structure of the scale:

The sequence:
power-up configuration $\longrightarrow$ Fixed $=$ `true` $\longrightarrow$ root $=$ E
results in



Then switching forth and back with
$\longrightarrow$ distance $=$ `fourths` $\longrightarrow$ distance $=$ `thirds`
with c_D as the fixpoint would result in



So we introduced c_R as the normal case, and take c_D only as a (rarely necessary) fall-back.

8.3.2 InKey $=$ `true`

In diatonic mode the changes to all parameters S, q, and d are problematic because nearly always the set of represented pitches changes—not all pads can keep their MIDI pitches. And (Q2) from the list of questions above (what is the "nearest pitch to C"?) must be answered.

On the other hand there are numerous *combinations* of changes which do not change the set of pitches, but only the ighting: Mixolydian on G holds the very same pitches as Lydian on F. Therefore we define a function change which can process a new scale and a new root in one step, and can be called with $q = q'$ or $S = S'$. (Especially a change of the root pitch but not the pitch class will have no effect at all!)

To leave as many pitches in place when changing root or scale, we search in every row from left to right for the first pad with a pitch present in both scale–root combinations. (P contains all new pitches, relative to the current root: add the new root to the new scale components to get MIDI pitches, and then subtract the old root to get relative indexes again. P' is the sequence of root-relative indexes the MIDI pitch of which will survive, and $w(r)$ is the first such index found in row r. Then $c(r)$ is the column number of that survivor, and the new start of the row $v'(r)$ is found $c(r)$ positions below $w(r)$ in the new scale.

Changing the scale in Fixed mode has always common pads, namely those of the root pitch class. Changing the root possibly has no common pad: Shift a whole-tone scale a half-tone up! In case $P' = \varnothing$ our implementation searches

// Universal types and data:

$$\mathsf{orientation} = \{\texttt{horizontal}, \texttt{vertical}\}$$
$$\mathsf{InKey} = \mathsf{boolean}$$
$$\mathsf{scale} = \mathbb{P}\,\{1..11\} \times \{1..11\}$$
$$\mathsf{distance} = \{\texttt{thirds}, \texttt{fourths}, \texttt{sequent}\}$$
$$\mathsf{roots} = \langle \mathsf{c}, \mathsf{c\sharp}, \mathsf{d}, \mathsf{d\sharp}, \mathsf{e}, \mathsf{f}, \mathsf{f\sharp}, \mathsf{g}, \mathsf{g\sharp}, \mathsf{a}, \mathsf{a\sharp}, \mathsf{b} \rangle$$
$$\mathsf{pitch} = (\mathsf{ran\ roots}) \times \mathbb{N}$$

// Calculate the key and lighting for one row:

$$\mathsf{role} = \{\texttt{finalis}, \texttt{repercussa}, \texttt{own}, \texttt{foreign}\}$$
$$sort : \mathbb{PN} \to \mathsf{seq}\,\mathbb{N}$$
$$sort(A) = squash(\mathbf{ID}_A)$$

$$\frac{S : \mathsf{scale} \qquad S' = \pi_1 S \cup \{0\} \cup \{\pi_2 S\} \qquad \widehat{S} = \bigcup_{o \in \mathbb{N}} (\_ + 12 * o)(\!|S'|\!)}{pitches(S) = sort(\widehat{S})}$$
$$pitches(S, s) = sort(\widehat{S} \cap \{s..\})$$

$$K_\mathrm{R} : \mathbb{N} \times \mathsf{boolean} \times \mathsf{scale} \times \mathsf{Column} \to \mathbb{N}$$
$$L_\mathrm{R} : \mathbb{N} \times \mathsf{boolean} \times \mathsf{scale} \times \mathsf{Column} \to \mathsf{role}$$
$$K_\mathrm{R}(s, \texttt{false}, \_, c) = s - 1 + c$$
$$K_\mathrm{R}(s, \texttt{true}, S, c) = pitches(S, s)(c)$$

$$\frac{p = K_\mathrm{R}(s, k, S, c)\ \mathsf{MOD}\ 12}{L_\mathrm{R}(s, k, S, c) = \begin{cases} \textbf{if } p = 0 & \textbf{then } \texttt{finalis} \\ \textbf{else if } p = \pi_2(S) & \textbf{then } \texttt{repercussa} \\ \textbf{else if } p \in \pi_1(S) & \textbf{then } \texttt{own} \\ \textbf{else} & \texttt{foreign} \end{cases}}$$

// Calculate the start of the next upper row:

$$N_\square : \mathbb{N} \times \mathsf{boolean} \times \mathsf{scale} \times \mathsf{distance} \to \mathbb{N}$$

$$N_\square(s, \texttt{false}, \_, \texttt{sequent}) = s + 8$$
$$N_\square(s, \texttt{true}, S, \texttt{sequent}) = pitches(S, s)(9)$$

$$N_\square(\_, \texttt{true}, S, \_) \in pitches(S)$$
$$s \in pitches(S) \wedge d \neq \texttt{sequent} \implies N_{1,3}(s, \texttt{false}, S, d) \in pitches(S)$$

$$N_1(s, \_, S, \texttt{thirds}) = pitches(S, s + 3)(1)$$
$$N_1(s, \_, S, \texttt{fourths}) = pitches(S, s + 5)(1)$$

$$N_2(s, \texttt{false}, \_, \texttt{fourths}) = s + 5$$

$$N_3(s, \_, S, \texttt{thirds}) = pitches(S, s)(3)$$
$$N_3(s, \_, S, \texttt{fourths}) = pitches(S, s)(4)$$

// Global state of the pitch layout:

$$V = \mathsf{Row} \to \mathbb{N}$$
$$\mathsf{initV} : \mathbb{N} \times \mathsf{boolean} \times \mathsf{scale} \times \mathsf{distance} \to V$$

$$\frac{v : V \quad r > 1 \qquad v(r) = N_\square(v(r-1), k, S, d) \qquad v(1) = c_0}{\mathsf{initV}(c_0, k, S, d) = v}$$

$$G = \mathsf{pitch} \times V \times \mathsf{boolean} \times \mathsf{scale} \times \mathsf{distance}$$
$$\mathsf{initG} : \mathsf{pitch} \times \mathsf{boolean} \times \mathsf{scale} \times \mathsf{distance} \to G$$
$$\mathsf{initG}(q, k, S, d) = (q, \mathsf{initV}(0, k, S, d), k, S, d)$$

Table 6. Pitch layout parameters and evaluation rules, part one.

// Look-up MIDI key values and lightimg:

$$[\![\_]\!] : \mathsf{pitch} \to \mathbb{N}$$
$$K : G \times \mathsf{Row} \times \mathsf{Column} \to \mathbb{N}$$
$$L : G \times \mathsf{Row} \times \mathsf{Column} \to \mathsf{role}$$

$$[\![(p, o)]\!] = \mathsf{roots}^{\sim}(p) - 1 + 12 * (o + 1)$$
$$K((q, v, k, S, d), r, c) = K_{\mathrm{R}}(v(r), k, S, c) + [\![q]\!]$$
$$L((\_, v, k, S, d), r, c) = L_{\mathrm{R}}(v(r), k, S, c)$$

// Dynamic changes in chromatic mode:

$$\mathsf{change}((q, v, \mathtt{false}, S, d), q') = (q', v \fatsemi (\lambda x \bullet x + q - q'), \mathtt{false}, S, d)$$
$$\mathsf{change}((q, v, \mathtt{false}, S, d), S') = (q, v, \mathtt{false}, S', d)$$

$$\frac{\begin{array}{c} c_{\mathrm{R}} = pitches((\varnothing, 0), v(1))(1) \\ c_{\mathrm{D}} = pitches(S, v(1))(1) \\ c_1 = \textbf{if } c_{\mathrm{R}} > v(1) + 7 \textbf{ then } c_{\mathrm{D}} \textbf{ else } c_{\mathrm{R}} \\ v' = \mathsf{initV}(c_1, \mathtt{false}, S, d') \\ v'' = v' \fatsemi (\lambda x \bullet x - c_1 + v(1)) \end{array}}{\mathsf{change}((q, v, \mathtt{false}, S, d), d') = (q, v'', \mathtt{false}, S, d')}$$

// Dynamic changes in diatonic mode:

$$\frac{\begin{array}{c} P = (\lambda x \bullet x - q + q')(\!|\mathsf{ran}\ pitches(S')|\!) \\ P' = pitches(S) \triangleright P \\ w(r) = squash(P' \triangleright \{v(r)..\})(1) \\ c(r) = pitches(S)^{\sim}(w(r)) - pitches(S)^{\sim}(v(r)) \\ v'(r) = pitches(S')\big(pitches(S')^{\sim}(w(r)) - c(r)\big) \\ v_1 = \min(P \cap \{v(1)..\}) + q - q' \end{array}}{\begin{array}{l} \mathsf{change}((q, v, \mathtt{true}, S, d), q', S') = \\ \begin{cases} \textbf{if } P` \neq \varnothing & \textbf{then } (q', v', \mathtt{true}, S', d) \\ \textbf{else} & (q', \mathsf{initV}(v_1, \mathtt{true}, S', d), \mathtt{true}, S', d) \end{cases} \end{array}}$$

$$\frac{v' = \mathsf{initV}(v(1), \mathtt{true}, S, d')}{\mathsf{change}((q, v, \mathtt{true}, S, d), d') = (q, v', \mathtt{true}, S, d')}$$

// Switching between chromatic and diatonic mode:

$$\frac{d \neq \mathtt{sequent}}{\mathsf{change}((q, v, \mathtt{true}, S, d), \mathtt{false}) = (q, v, \mathtt{false}, S, d)}$$

$$\frac{d = \mathtt{sequent} \quad v' = \mathsf{initV}(v(1), \mathtt{false}, S, d)}{\mathsf{change}((q, v, \mathtt{true}, S, d), \mathtt{false}) = (q, v', \mathtt{false}, S, d)}$$

$$\frac{v' = v \fatsemi (\lambda x \bullet pitches(S, x)(1))}{\mathsf{change}((q, v, \mathtt{false}, S, d), \mathtt{true}) = (q, v', \mathtt{true}, S, d)}$$

Table 7. Pitch layout parameters and evaluation rules, part two.

for the minimal new pitch which is higher than the current pitch at row one, column one, and creates from there an initial layout.

So the "nearest note to C" is the note which leaves as many pads as possible unchanged. If there are no common pitches between both pitch layouts, then it is the next higher pitch above C in the new layout. This second part of the rule is totally arbitrary and will not give stable results when applied repeatedly!

When distance changes, we construct a new pitch layout but keep $v(1)$ and thus the complete first row. All other rows must change necessarily.

8.3.3 Changing InKey

The set of pitches always changes when altering InKey, which is switching between chromatic and diatonic mode. According to the "principle of least change", the proposal in Table 7 keeps the starting points of all rows as far as possible: From chromatic to diatonic it is a *compaction*, where the first pad in every row becomes the first diatonic pad in the preceding layout.

From diatonic to chromatic the row starts are kept completely.

This is not a solution for distance = `sequent`: The preceding layout

…							
D4	E4	F4	G4	A4	B4	**C5**	D5
C3	D3	E3	F3	G3	A3	B3	**C4**

would be changed to

…							
D4	D♯4	**E4**	**F4**	F♯4	**G4**	G♯4	**A4**
C3	C♯3	**D3**	D♯3	**E3**	**F3**	F♯3	**G3**

and thus effectively *lose access* to diatonic pitches. In this case the fresh construction of an initial layout starting with the pitch in pad $(1, 1)$ seems more appropriate.

8.3.4 Conclusion

The sheer number of formula lines needed to describe a possible solution shows that a precise specification cannot be expected from just one manual page in prose language.

9. IMPLEMENTATION

We provide an implementation of our slightly extended version of TAB+, Chart+, the `tScore` representation, the data model,, and the pitch layout algorithm in one Java application. It can be downloaded from `http://bandm.eu/downloads/SquarePads.jar`. It can be started from the command line (CLI) as usual, taking options as explained by the option "`--help`", or by simply clicking the jar file icon, which requires to enter option values through its GUI.

Without any `tScore` file input, the application can be used to experiment with the different flavours of the pitch layout algorithm: The above-mentioned parameters InKey, scale, distance, base pitch, and Fixed can be changed dynamically, using the GUI widgets. The effects are shown on the GUI grid, see Figure 8. On this grid, simple and monophonic sounds can be played by clicking, produced by self-contained sound synthesis.

Additionally a hardware can be connected, on which the lighting is also shown on the pads, and the sounds can be played by pressing them. Currently we support only the "Novation Launchpad MK3" TUI device—probably adapting the mapping of the MIDI commands will be necessary for other hardware.

Selecting a file which contains a `tScore` source allows to render it into a sequence of SVG files containing the

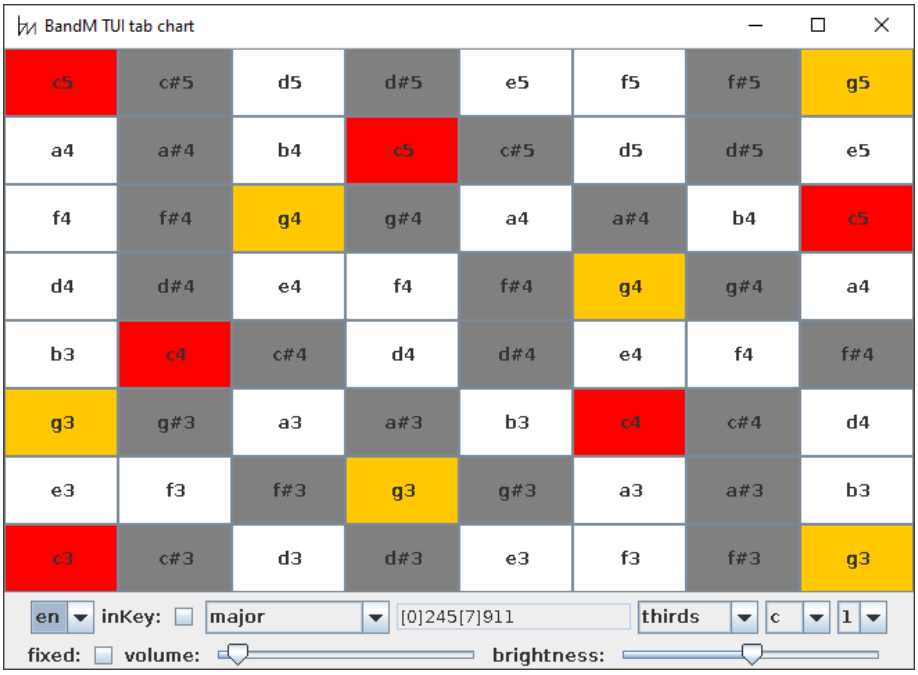


Figure 8. Screenshot of the tool in interactive mode

topic	lines of code
`tScore` model	335
chart rendering	277 + 148
tablature rendering and GUI	323 + 217
dynamic change of pitch layout	860

Table 8. Lines of code of the current implementation. (Measured with CLoC[16])

sequence of charts, into one file with an animated chart, and into tablature output (currently only one measure). By using CLI this can be done in batch mode. Activating interactive mode allows sequencing the score auditive, visually on the GUI and the connected hardware, and to a MIDI-out channel.

The Java sources are licensed as CC-BY-SA-NC and published on a source hosting platform. Table 8 shows the surprisingly low programming effort. The tool uses BandM metatools, metricSplit [12, 13], and Sig [14]. It uses the JSVG library by Jonathan Sevy [15].

10. SUPPORT FOR INTERDISCIPLINARY RESEARCH AND PRACTICE

In principle, all three notation formats defined by Wilde and White can be applied to any input device which has an orthogonal grid structure of binary-state input buttons, with arbitrary (manageable) numbers of rows and columns.

Since MIDI events can be translated into any kind of data, such TUIs and touchscreen GUIs can be used as input device for any interactive software, e.g. in psychological experiments, on-line examinations, or as a game controller. (We intend to explore them as a playing device for a synthesizer with pure intonation, mapping the *Euler net* to the two axes. To support TUIs of other vendors, the existing software must be enhanced by an adaption layer for different usage of MIDI commands.)

All formats presented here, Charts+, TAB+, their data model, and the `tScore` format, deal only with the physical coordinates of the buttons. Therefore they can be employed in all these cases for the pressed keys and the pressing fingers. They can be used for protocols of user interaction, or as prescribing score notation for human or automated execution. Translating the notation into the data model allows statistical analyses, and to apply properties (like those defined above in Table 3) for consistency checks.

There is one limitation: TAB+, as presented by Wilde and White, is restricted to cases which employ conventional meter-based rhythmical timing, and to users which are familiar with the duration encoding by CWN. Contrarily, the `tScore` representation can easily be adopted to other time models, e.g. physical time, or mere sequential order without any duration, and so can our rendering algorithm for tablatures. This opens a wider spectrum of applicability.

The other way of translation still needs exploring: creating timed sequences of pad patterns from traditional note information. This is a challenge because the physiological constraints of the human hands must be respected, see for instance the data type PressPattern in Table 3.

11. ACKNOWLEDGEMENTS

Many thanks to Lawrence Wilde and Charles White for additional information about their work.

Many thanks to Jon Sevy for realizing our wishes in his JSVG library.

Many thanks to the anonymous reviewers; not all of their valuable hints could be realized, due to lack of time.

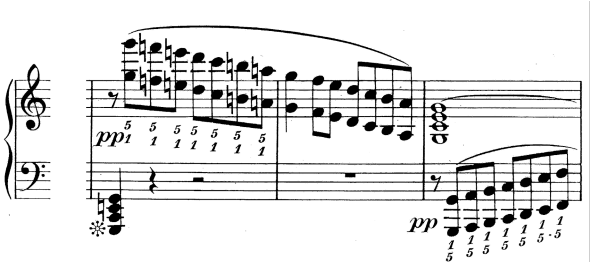


Figure 9. Finger numbers in the Waldstein sonata

A. IRREGULARITIES HERE AND IN CONVENTIONAL KEYBOARD CWN

Often and in very different realms, the authors have experienced that the attempt to construct a mathematical (meta-)model brings new insights into the pre-digital historic practice, especially bringing to light non-orthogonalities and idiosyncrasies. Here, modelling the TUI can be seen as preparatory for modelling the playing on a conventional chromatic keyboard.

Some irregular and noteworthy aspects:

(A) At the beginning (of the conventional CWN text) of a piece (for piano, organ, etc.) there is nearly always an implicit assignment of the notes to the two hands, by the implicit meaning of upper and lower staff.

Later, an explicit prescription of left or right hand can appear by the textual *m.s.* annotation. This is a *sticky* attribute:[5] It is written over one note and *stays valid* for some subsequent notes.

But there is no notational counterpart meaning "the prescription ends here" or "from here on the hands are free again", like a *loco* instruction after an *8va sopra*.[6]

(B) The pragmatic aspect, as linguistics call it, of these annotations can be different, namely a prescription, meant as an indispensable part of the work, or just a recommendation. When authored by the composer and not the editor, both are conceivable.

(C) Statements (A) and (B) also hold for finger numbers. Especially it is unclear whether such a number holds only for the attack ("push down with this finger, afterwards you may silently change") or for the whole notated duration ("push down with this finger and stick to it"). In other words: In conventional CWN there are signs to prescribe a silent finger change, but not to forbid it.

Similar to (A), there is no way to notate "the prescribed fingering ends here", for instance at one notehead in a long tied sequence, For this purpose one could learn from the `tScore` implementation above the sequence "`12R3␣␣>R`" and translate it into the conventional "3␣␣*m.d.*".

When humans play piano, these questions are practically not an issue. But they *must* be answered when implementing an algorithmic simulation, an automated interpretation by a robot, etc.

(D) In conventional CWN for keyboard, a *finger indication* is only sensible when the *hand* is also prescribed. The only exception we know of are the (in)famous octave glissandi in the last movement of the Waldstein sonata, as proposed by Bertha Wallner, [17] see Figure 9: $\frac{1}{5}$ *could* be read as "perform (as a glissando) with small finger and thumb of the same hand, right or left as you like".[7]

(E) In conventional CWN for keyboard, a *finger indication* is only sensible when a key is pressed. But a *hand indication* can also be sensible with a *pause*! The information "this finger does currently not play" is normally not made explicit in conventional notation, but "this hand pauses" is! Especially for rendering in traditional piano staff notation, any syntactic entity which is a pause needs the "which hand" information, because this normally decides in which of the two staves to put the representing glyph.

[5] Called NOTA.ADDENDAMANENTIA in LMN [8].

[6] Called *Aufhebungsereignis* = PERMANENS.EXIT in LMN [8].

[7] Again the status of this statement is critical anyhow: It is an information how these notes *had been* executed on the instruments of that time, and not a hint how to perform today on modern instruments.

B. REFERENCES

[1] L. Wilde and C. White, "TABstaff+: A hybrid music notation system for grid-based tangible user interfaces (tuis) and graphical user interfaces (guis)," in *Tenor 2024 proceedings*, Zürich, 2024, pp. 159–168, https://www.tenor-conference.org/proceedings/TENOR2024-Proceedings.pdf.

[2] C. Sperren, *A notation system for Ableton Push*, 2020, https://www.chrisperren.com/blog/2020/12/02/a-notation-system-for-ableton-push[20240724].

[3] S. Riegel, A. Kuźbik, M. Hughes, C. Tipton, J. Evans, and J. Terry, *Ableton Push 3 Manual*, Berlin, 2020, https://www.ableton.com/en/push/manual/[20250617].

[4] M. Lepper and B. Trancón y Widemann, "tscore: Makes computers and humans talk about time," in *Proc. KEOD 2013, 5th Intl. Conf. on Knowledge Engineering and Ontology Development*, J. Filipe and J. Dietz, Eds., instincc. Portugal: scitePress, 2013, pp. 176–183, http://markuslepper.eu/papers/tscore2013.pdf[20231003].

[5] ——, "Translets —— parsing diagnosis in small dsls, with permutation combinator and epsilon productions," in *Tagungsband des 35ten Jahrestreffens der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte*, ser. IFI Reports, vol. 482. University of Oslo, 2018, pp. 114–129, http://urn.nb.no/URN:NBN:no-65294[20230920].

[6] ——, "Morton Feldman's "projections one to five" — exploring a classical avant-garde notation by mathematical remodelling," in *Proceedings of TENOR 2024 conference*, 2024, https://www.tenor-conference.org/proceedings/2024/15_TENOR2024_Lepper.pdf[20250701].

[7] ——, *Example Instances of the TScore Project Infrastructure*, 2013, http://markuslepper.eu/sempart/tscoreInstances.html[20230920].

[8] M. Lepper, "De linguis musicam notare," Ph.D. dissertation, Universität Osnabrück, Osnabrück, 2021, https://www.epos.uni-osnabrueck.de/buch.html?id=150[20230920].

[9] J. Wolf, *Handbuch der Notationskunde, Teil zwei*. Leipzig: Breitkopf & Härtel, 1919, https://ia800205.us.archive.org/3/items/handbuchdernotat02wolf/handbuchdernotat02wolf.pdf[20230101].

[10] C. Pot, *Klavarskribo und die Tontheorie*, 1932.

[11] J. Oberschmidt, "Zwischen Mensch und Maschine. Komponieren für Piano-Player und Player Piano," in *Musik im Spektrum technologischer Entwicklungen und Neuer Medien*. Osnabrück: epOs Verlag, 2015, p. 563–580.

[12] M. Lepper and B. Trancón y Widemann, "Metricsplit - Automated Notation of Rhythms, Adequate to a Metric Structure," Technische Universität Ilmenau, Ilmenau, Tech. Rep., Mar 2017.

[13] ——, "metricsplit: An automated notation of rhythm aligned with metric structure," *Journal of Mathematics and Music*, vol. 13, no. 1, pp. 60–84, 2019.

[14] B. Trancón y Widemann and M. Lepper, "Sig adlib — mostly compositional clocked synchronous data-flow programming in java," in *Tagungsband des 35ten Jahrestreffens der GI-Fachgruppe Programmiersprachen und Rechenkonzepte*, ser. IFI Reports, vol. 482. University of Oslo, 2018, pp. 31–48. [Online]. Available: http://urn.nb.no/URN:NBN:no-65294

[15] J. Sevy, *JSVG:Java SVG library*, 2023, https://jsevy.com/wordpress/index.php/java-and-android/java-libraries/jsvg-java-svg-library/[20250708].

[16] A. Danial, "cloc — count lines of code (v1.92)," Tech. Rep., 2021, https://github.com/AlDanial/cloc [20231003]. [Online]. Available: https://doi.org/10.5281/zenodo.5760077

[17] L. van Beethoven, *Klaviersonaten II*. München: Henle Verlag, 1952, *edt.* Bertha Antonia Wallner.

[18] J. M. Spivey, *The Z Notation: A reference manual*, ser. International Series in Computer Science. Prentice Hall, 1988, https://spivey.oriel.ox.ac.uk/corner/Z_Reference_Manual[20240729].

C. MATHEMATICAL NOTATION

The employed mathematical notation is fairly standard, inspired by the Z notation [18]. The following table lists some details:

$\mathbb{N}$	All natural numbers, incl. zero.
pred	The predecessor operation on integer numbers.
$\#A$	The cardinality of a finite set = the natural number of the elements contained.
$\mathbb{P}A$	Power set, the type of all subsets of the set A, incl. infinites.
OPT A	The optional type of all values from A plus the undefined value $\perp$. (Our extension; could be realized as a generic schema.)
$\varnothing$	The empty set (a constant of any type)
$A \setminus B$	The set containing all elements of A which are not in B.
$A \times B$	The product type of two sets A and B, i.e. all pairs $\{c = (a, b) \mid a \in A \wedge b \in B\}$.
π_n	The nth component of a tuple.
$A \to B$	The type of the *total* functions from A to B.
$A \nrightarrow B$	The type of the *partial* functions from A to B.
$f \llparenthesis s \rrparenthesis$	The image of the set s under the function or relation f.
$a \mapsto b$	An element of a relation; simply another way to write (a, b).
dom A	Domain of a relation or function.
ran A	Range of a relation or function.
$r^\sim$	The inverse of a relation or function.
$\mathbf{ID}_A$	$= \{a \in A \bullet a \mapsto a\}$, the identity relation.
$r \fatsemi s$	The composition of two relations: the smallest relation s.t. $a\ r\ b \wedge b\ s\ c \Rightarrow a\ (r \fatsemi s)\ c$. (first apply r, then apply s.)
$r \oplus s$	Overriding of function or relation r by s. Pairs from r are shadowed by pairs from s: $r \oplus s = \big(r \setminus (\mathsf{dom}\ s \times \mathsf{ran}\ r)\big) \cup s$, with dom and ran being domain and range, resp.
$S \lhd R$	$= R \cap (S \times \mathsf{ran}\ R)$, i.e. domain restriction of a relation.
$R \rhd S$	$= R \cap (\mathsf{dom}\ R \times S)$, i.e. range restriction of a relation.
$R \rhd\!\!\!\!\!\!\cdot\ S$	$= R \setminus (\mathsf{dom}\ R \times S)$, i.e. negative range restriction of a relation.
seq A	The type of finite sequences from elements of A, i.e. of maps $\mathbb{N} \nrightarrow A$ with a contiguous range $\{1..n\}$ as its domain.
$squash(a)$	Turns any partial function $\mathbb{N} \nrightarrow A$ into a seq A by compactifying the indices.
$[\ expr\ ]$	Iverson bracket: delivers 1 if the boolean expression evaluates to true, and 0 if to false. (our extension)
max A	The maximum from a set of values.
min A	The minimum from a set of values.

The frequently used notation

$$\frac{\begin{array}{cc} & a \\ b & \quad c \end{array}}{d}$$

means as usual $a \wedge b \wedge c \implies d$. Nearly always it should be read as an *algorithm*: "For to calculate d, try to calculate a, b and c. If this succeeds, the answer d is valid."

Functions are considered as special relations; relations as sets of pairs. So with functions, expressions like "$f \cup g$" are defined.