Scanning methods and language modeling for binary switch typing

Brian Roark[†], Jacques de Villiers[†], Christopher Gibbons[°] and Melanie Fried-Oken[°]
[†]Center for Spoken Language Understanding [°]Child Development & Rehabilitation Center
Oregon Health & Science University
{roark,jacques}@cslu.ogi.edu {gibbons,mfo}@ohsu.edu

Abstract

We present preliminary experiments of a binary-switch, static-grid typing interface making use of varying language model contributions. Our motivation is to quantify the degree to which language models can make the simplest scanning interfaces – such as showing one symbol at a time rather than a scanning a grid – competitive in terms of typing speed. We present a grid scanning method making use of optimal Huffman binary codes, and demonstrate the impact of higher order language models on its performance. We also investigate the scanning methods of highlighting just one cell in a grid at any given time or showing one symbol at a time without a grid, and show that they yield commensurate performance when using higher order n-gram models, mainly due to lower error rate and a lower rate of missed targets.

1 Introduction

Augmentative and Alternative Communication (AAC) is a well-defined subfield of assistive technology, focused on methods that assist individuals for whom conventional spoken or written communication approaches are difficult or impossible. Those who cannot make use of standard keyboards for text entry have a number of alternative text entry methods that permit typing. One of the most common of these alternative text entry methods is the use of a binary switch – triggered by button-press, eye-blink or even through event related potentials (ERP) such as the P300 detected in EEG signals – that allows the individual to make a selection based on some method for scanning through alternatives (Lesher et al., 1998). Typing speed is a challenge, yet critically important for usability. One common approach is row/column scanning on a matrix of characters, symbols or images (a 'spelling grid'), which allows the user of a binary yes/no switch to select the row and column of a target symbol, by simply indicating 'yes' (pressing a button or blinking an eye) when the row or column of the target symbol is highlighted. Figure 1 shows the 6×6 spelling grid used for the P300 Speller (Farwell and Donchin, 1988).

For any given scanning method, the use of a binary switch to select from among a set of options (letter, symbols, or images) amounts to the assignment of binary codes to each symbol. For example, the standard row/column scanning algorithm works by scanning each row until a selection is made, then scanning each column until a selection is made, and returning the symbol at the selected row and column. This can be formalized as follows:

```
1   for i = 1 to (# of rows) do
2       HIGHLIGHTROW(i)
3       if YESSWITCH
4           for j = 1 to (# of columns) do
5               HIGHLIGHTCOLUMN(j)
6               if YESSWITCH
7                   return (i, j)
8           return (i, 0)
9   return (0, 0)
```

where the function YESSWITCH returns *true* if the button is pressed (or whatever switch event counts as a 'yes' response) within the parameterized latency. If the function returns $(0,0)$ then nothing has been selected, requiring rescanning. If the function returns $(i,0)$ for $i > 0$, then row i has been selected, but columns must be rescanned. Under this scanning method, the binary code for the letter 'J' in the matrix in Figure 1 is 010001; the letter 'T' is 000101.

The length of the binary code for a symbol is re-

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R
S	T	U	V	W	X
Y	Z	1	2	3	4
5	6	7	8	9	_

Figure 1: Spelling grid such as that used for the P300 speller (Farwell and Donchin, 1988). '_' denotes space.

lated to the time required to type it. In the matrix in Figure 1, the space character is in the bottom right-hand corner, yielding the maximum binary code length for that grid size (12), despite that, in typical written English we would expect the space character to be used about 20% of the time. A more efficient strategy would be to place the space character in the upper left-hand corner of the grid, leading to the much shorter binary code '11'.

Ordering symbols in a fixed grid so that frequent symbols are located in the upper left-hand corner is one method for making use of a statistical model of the language so that likely symbols receive the shortest codes. Such a language model, however, does not take into account what has already been typed, but rather assigns its code identically in all contexts. In this paper we examine alternative fixed-grid scanning methods that do take into account context in the language models used to establish codes, i.e., the codes in these methods vary in different contexts, so that high probability symbols receive the shortest codes and hence require the fewest keystrokes. We show that n-gram language models can provide a large improvement in typing speed.

Before presenting our methods and experimental results, we next provide further background on alternative text entry methods, language modeling, and binary coding based on language models.

2 Preliminaries and background
2.1 Alternative text entry
Of the ways in which AAC typing interfaces differ, perhaps most relevant to the current paper is whether the symbol positions are fixed or can move dynamically, because such dynamic layouts facilitate integration of richer language models. For example, if we re-calculate character probabilities after each typed character, then we could re-arrange the characters in the grid so that the most likely are placed in the upper left-hand corner for row/column scanning. Conventional wisdom, however, is that the cognitive overhead of processing a different grid arrangement after every character would slow down typing more than the speedup due to the improved binary coding (Baletsa et al., 1976; Lesher et al., 1998). The GazeTalk system (Hansen et al., 2003), which presents the user with a 3×4 grid and captures which cell the user's gaze fixates upon, is an instance of a dynamically changing grid. The cell layouts are configurable, but typically one cell contains a set of likely word completions; others are allocated to space and backspace; and around half of the cells are

allocated to the most likely single character continuation of the input string, based on language model predictions. Hansen et al. (2003) report that users produced more words per minute with a static keyboard than with the predictive grid interface, illustrating the impact of the cognitive overhead that goes along with this sort of scanning.

The likely word completions in the GazeTalk system illustrates another common way in which language modeling is integrated into AAC typing systems. Much of the language modeling research within the context of AAC has been for word completion/prediction for keystroke reduction (Darragh et al., 1990; Li and Hirst, 2005; Trost et al., 2005; Trnka et al., 2006; Trnka et al., 2007; Wandmacher and Antoine, 2007). The typical scenario for this is allocating a region of the interface to contain a set of suggested words that complete what the user has begun typing. The expectation is to derive a keystroke savings when the user selects one of the alternatives rather than typing the rest of the letters. The cognitive load of monitoring a list of possible completions has made the claim that this speeds typing controversial (Anson et al., 2004); yet some results have shown this to speed typing under certain conditions (Trnka et al., 2007).

One innovative language-model-driven AAC typing interface is Dasher (Ward et al., 2002), which uses language models and arithmetic coding to present alternative letter targets on the screen with size relative to their likelihood given the history. Users can type by continuous motion, such as eye gaze or mouse cursor movement, targeting their cursor at the intended letter and moving the cursor from left-to-right through the interface, while its movements are tracked. This is an extremely effective typing interface alternative to keyboards, provided the user has sufficient motor control to perform the required systematic visual scanning. The most severely impaired users, such as those with locked-in syndrome (LIS), have lost the voluntary motor control sufficient for such an interface.

Relying on extensive visual scanning, such as that required in dynamically reconfiguring spelling grids or Dasher, or requiring complex gestural feedback from the user renders a typing interface difficult or impossible to use for those with the most severe impairments. Indeed, even spelling grids like the P300 speller can be taxing as an interface for users. Recent attempts to use the P300 speller as a typing interface for locked-in individuals with ALS found

```
1   A ← V      ▷ initialize A as symbol set V
2   k ← 1      ▷ initialize bit position k to 1
3   while |A| > 1 do
4          P ← {a ∈ A : a[k] = 1}
5          Q ← {a ∈ A : a[k] = 0}
6          Highlight symbols in P
7          if selected then  A ← P
8          else  A ← Q
9          k ← k + 1
10  return a ∈ A      ▷ Only 1 element in A
```

Figure 2: Algorithm for binary code symbol selection

| Letter:	a	b	c	d	e	f
Probability:	0.15	0.25	0.18	0.2	0.12	0.1
Huffman bits:	3	2	3	2	3	3
Linear bits:	4	1	3	2	5	5

Figure 3: Two binary trees for encoding letters based on letter probabilities: (1) Huffman coding; and (2) Linear coding via a right-branching tree (right-linear). Expected bits are 2.55 for Huffman and 2.89 for linear coding.

that the number of items in the grid caused problems for these patients, because of difficulty orienting attention to specific locations in the spelling grid (Sellers et al., 2003). This is another illustration of the need to reduce the cognitive overhead of such interfaces. Yet the success of classification of ERP in a simpler task for this population indicates that the P300 is a binary response mechanism of utility for this task (Sellers and Donchin, 2006).

Simpler interactions via brain-computer interfaces (BCI) hold much promise for effective text communication. Yet these simple interfaces have yet to take full advantage of language models to ease or speed typing. In this paper we will make use of a static grid, or a single letter linear scanning interface, yet scan in a way that allows for the use of contextual language model probabilities when constructing the binary code for each symbol.

2.2 Binary codes for typing interfaces

Row/column scanning, as outlined in the previous section, is not the only means by which the spelling grid in Figure 1 can be used as a binary response typing interface. Rather than highlighting full rows or full columns, arbitrary subsets of letters could be highlighted, and letter selection again driven by a binary response mechanism. An algorithm to do this is as follows. Assign a unique binary code to each symbol in the symbol set V (letters in this case). For each symbol $a \in V$, there are $|a|$ bits in the code representing the letter. Let $a[k]$ be the kth bit of the code for symbol a. We will assume that no symbol's binary code is a prefix of another symbol's binary code. Given such an assignment of binary codes to the symbol set V, the algorithm in Figure 2 can be used to select the target symbol in a spelling grid.

One key question in this paper is how to produce such a binary code, which is how language models can be included in scanning. Figure 3 shows two different binary trees, which yield different binary codes for six letters in a simple, artificial example.
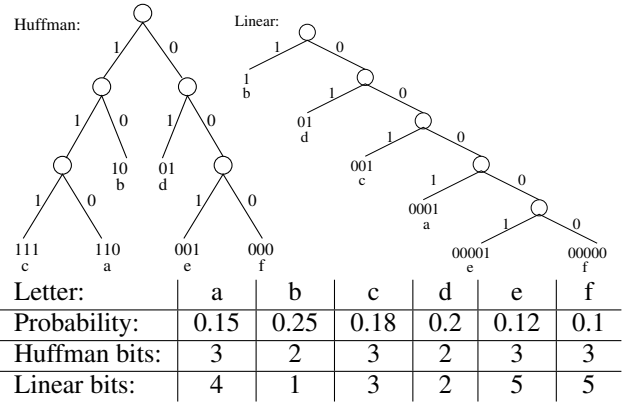
Huffman coding (Huffman, 1952) builds a binary tree that minimizes the expected number of bits according to the provided distribution. There is a linear complexity algorithm for building this tree given a list of items sorted by descending probability.

Another type of binary code, which we will call a linear code, provides a lot of flexibility in the kind of interface that it allows, relative to the other methods mentioned above. In this binary code, each iteration of the WHILE loop in the Figure 2 algorithm would have a set P on line 4 with exactly one member. With such a code, the spelling grid in Figure 1 would highlight exactly one letter at a time for selection. Alternately, symbols could be presented one at a time with no grid, which we call rapid serial visual presentation (RSVP, see Fig.7). Linear coding builds a simple right-linear tree (seen in Figure 3) that preserves the sorted order of the set, putting higher probability symbols closer to the root of the tree, thus obtaining shorter binary codes. Linear coding can never produce codes with fewer expected bits than Huffman coding, though the linear code may reach the minimum under certain conditions.

The simplicity of an interface that presents a single letter at a time may reduce user fatigue, and even make typing feasible for users that cannot maintain focus on a spelling grid. Additionally, single symbol auditory presentation would be possible, for visually impaired individuals, something that is not straightforwardly feasible with the sets of symbols that must be presented when using Huffman codes.

2.3 Language modeling for typing interfaces

The current task is very similar to word prediction work discussed in Section 2.1, except that the pre-

diction interface is the only means by which text is input, rather than a separate window with completions being provided. In principle, the symbols that are being predicted (hence typed) can be from a vocabulary that includes multiple symbol strings such as words. However, a key requirement in a composition-based typing interface is an **open vocabulary** – the user should be able to type any word, whether or not it is in some fixed vocabulary. Included in such a mechanism is the ability to repair: delete symbols and re-type new ones. In contrast, a word prediction component must be accompanied by some additional mechanism in place for typing words not in the vocabulary. The current problem is to use symbol prediction for that core typing interface, and this paper will focus on predicting single ASCII and control characters, rather than multiple character strings. The task is actually very similar to the well known Shannon game (Shannon, 1950), where text is guessed one character at a time.

Character prediction is done in the Dasher and GazeTalk interfaces, as discussed in an earlier section. There is also a letter prediction component to the Sibyl/Sibylle interfaces (Schadle, 2004; Wandmacher et al., 2008), alongside a separate word prediction component. Interestingly, the letter prediction component of Sibylle (Sibyletter) involves a linear scan of the letters, one at a time in order of probability (as determined by a 5-gram character language model), rather than a row/column scanning of the P300 speller. This approach was based on user feedback that the row/column scanning was a much more tiring interface than the linear scan interface (Wandmacher et al., 2008), which is consistent with the results previously discussed on the difficulty of ALS individuals with the P300 speller interface.

Language modeling for a typing interface task of this sort is very different from other common language modeling tasks. This is because, at each symbol in the string, the already typed prefix string is given – there is no ambiguity in the prefix string, modulo subsequent repairs. In contrast, in speech recognition, machine translation, optical character recognition or T9 style text input, the actual prefix string is not known; rather, there is a distribution over possible prefix strings, and a global inference procedure is required to find the best string as a whole. For typing, once the symbol has been produced and not repaired, the model predicting the next symbol is given the true context. This has several important ramifications for language modeling,

including the availability of supervised adaptation data and the fact that the models trained with relative frequency estimation are both generative and discriminative. See Roark (2009) for extensive discussion of these issues. Here we will consider n-gram language models of various orders, estimated via smoothed relative frequency estimation (see § 3.1). The principal novelty in the current approach is the principled incorporation of error probabilities into the binary coding approaches, and the experimental demonstration of how linear coding for grids or RSVP interfaces compare to Huffman coding and row/column scanning for grids.

3 Methods

3.1 Character-based language models

For this paper, we use character n-gram models. Carpenter (2005) has an extensive comparison of large scale character-based language models, and we adopt smoothing methods from that paper. It presents a version of Witten-Bell smoothing (Witten and Bell, 1991) with an optimized hyperparameter K, which is shown to be as effective as Kneser-Ney smoothing (Kneser and Ney, 1995) for higher order n-grams. We refer readers to that paper for details on this standard n-gram language modeling approach. For the experimental results presented here, we trained unigram and 8-gram models from the NY Times portion of the English Gigaword corpus.

We performed extensive normalization of this corpus, detailed in Roark (2009). We de-cased the resulting corpus and selected sentences that only included characters that would appear in our $6{\times}6$ spelling grid. Those characters are: the 26 letters of the English alphabet, the space character, a delete symbol, comma, period, double and single quote, dash, dollar sign, colon and semi-colon. We used a 42 million character subset of this corpus for training the model. Finally, we appended to this corpus approximately 112 thousand words from the CMU Pronouncing Dictionary (`www.speech.cs.cmu.edu/cgi-bin/cmudict`), which also contained only the symbols from the grid. For hyper-parameter settings, we used a 100k character development set. Our best performing hyper-parameter for the Witten-Bell smoothing was $K = 15$, which is comparable to optimal settings found by Carpenter (2005) for 12-grams.

3.2 Binary codes

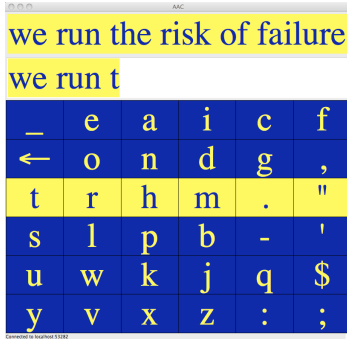Given what has been typed so far, we can use a character n-gram language model to assign probabilities

Figure 4: Row/column scanning interface.

Figure 5: Error in row/column scanning interface.

to all next symbols in the symbol set V. After sorting the set in order of decreasing probability, we can use these probabilities to build binary coding trees for the set. Hence the binary code assigned to each symbol in the symbol set differs depending on what has been typed before. For Huffman coding, we used the algorithm from Perelmouter and Birbaumer (2000) that accounts for any probability of error in following a branch of the tree, and builds the optimal coding tree even when there is non-zero probability of taking a branch in error. Either linear or Huffman codes can be built from the language model probabilities, and can then be used for a typing interface, using the algorithm presented in Figure 2.

3.3 Scanning systems

For these experiments, we developed an interface for controlled testing of typing performance under a range of scanning methods. These include: (i) row/column scanning, both auto scan (button press selects) and step scan (lack of button press selects); (ii) Scanning with a Huffman code, either derived from a unigram language model, or from an 8-gram language model; and (iii) Scanning with a linear code, either on the 6×6 grid, or using RSVP, which shows one symbol at a time. Each trial involved giving subjects a target phrase with instructions to type the phrase exactly as displayed. All errors in typing were required to be corrected by deleting (via ←) the incorrect symbol and re-typing the correct symbol.

Figure 4 shows our typing interface when configured for row/column scanning. At the top of the application window is the target string to be typed by the subject ('we run the risk of failure'). Below that is the buffer displaying what has already been typed ('we run t'). Spaces between words must also be typed – they are represented by the underscore character in the upper left-hand corner of the grid. Spaces are treated like any other symbol in our language model – they must be typed, thus they are pre-
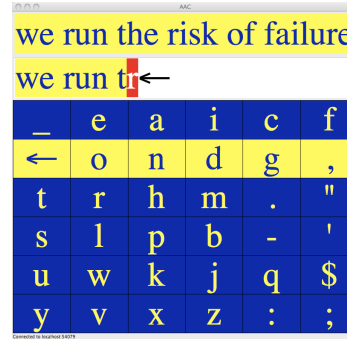
dicted along with the other symbols. Figure 5 shows how the display updates when an incorrect character is typed. The errors are highlighted in red, followed by the backarrow symbol to remind users to delete.

If a row has not been selected after a pass over all rows, scanning begins again at the top. After row selection, column scanning commences; if a column is not selected after three passes from left-to-right over the columns, then row scanning re-commences at the following row. Hence, even if a wrong row is selected, the correct symbol can still be typed.

Note that the spelling grid has been sorted in unigram frequency order, so that the most frequent symbols are in the upper left-hand corner. This same grid is used in all grid scanning conditions, and provides language modeling benefit to row/column scanning.

Figure 6 shows our typing interface when configured for what we term Huffman scanning. In this scanning mode, the highlighted subset is dictated by the Huffman code, and is not necessarily contiguous. Not requiring contiguity of highlighted symbols allows the coding to vary with the context, thus allowing use of an n-gram language model. As far as we know, this is the first time that contiguity of highlighting is relaxed in a scanning interface to accommodate Huffman coding. Baljko and Tam (2006) used Huffman coding for a grid scanning interface, but using a unigram model and the grid layout was selected to ensure that highlighted regions would always be contiguous, thus precluding n-gram models.

In our Huffman scanning approach, when the selected set includes just one character, it is typed. As with row/column scanning, when the wrong character is typed, the backarrow symbol must be chosen to delete it. If an error is made in selection that does not result in a typed character – i.e., if the incorrectly selected set has more than one member – then we need some mechanism for allowing the target symbol to still be selected, much as we have a mecha-

Figure 6: Huffman scanning interface.

Figure 7: RSVP scanning interface.

nism in row/column scanning for recovering if the wrong row is selected. Section 3.4 details our novel method for recalculating the binary codes based on an error rate parameter. At no point in typing is any character ruled out from being selected.

The grids shown in Figures 4-6 can be straightforwardly used with linear coding as well, by simply highlighting one cell at a time in descending probability order. Additionally, linear coding can be used with an RSVP interface, shown in Figure 7, which displays one character at a time.

Each interface needs a scan rate, specifying how long to wait for a button press before advancing. The scan rate for each condition was set for each individual during a training/calibration session (see §4.1).

3.4 Errors in Huffman and Linear scanning

In this section we briefly detail how we account for the probability of error in scanning with Huffman and linear codes. The scanning interface takes a parameter p, which is the probability that, when a selection is made, it is correct. Thus $1-p$ is the probability of an error. Recall that if a selection leads to a single symbol, then that symbol is typed. Otherwise, if a selection leads to a set with more than one symbol, then *all* symbol probabilities (even those not in the selected set) are updated based on the error probability and scanning continues. If a non-target (incorrect) symbol is selected, the delete (backarrow) symbol must be chosen to correct the error, after which the typing interface returns to the previous position. Three key questions must be answered in such an approach: (1) how are symbol probabilities updated after a keystroke, to reflect the probability of error? (2) how is the probability of backarrow estimated? and (3) when the typing interface returns to the previous position, where does it pick up the scanning? Here we answer all three questions.

Consider the Huffman coding tree in Figure 3. If the left-branch ('1') is selected by the user, the probability that it wa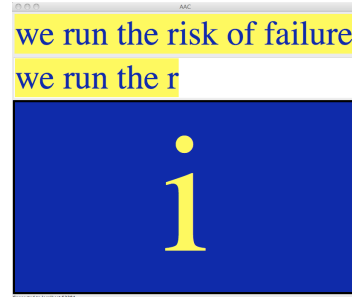s intended is p versus an error with probability $1-p$. If the original probability of a symbol is q, then the updated probability of the symbol is pq if it starts with a '1' and $(1-p)q$ if it starts with a '0'. After updating the scores and re-normalizing over the whole set, we can build a new binary coding tree. The user then selects a branch at the **root** of the new tree. A symbol is finally selected when the user selects a branch leading to a single symbol. The same approach is used with a linear coding tree.

The probability of requiring the delete (backarrow) character can be calculated directly from the probability of keystroke error – in fact, the probability of backarrow is exactly the probability of error $1-p$. To understand why this is the case, consider that a non-target (incorrect) symbol can be chosen according to the approach in the previous paragraph only with a final keystroke error. Any keystroke error that does not select a single symbol does not eliminate the target symbol, it merely re-adjusts the target symbol's probability along with all other symbols. Hence, no matter how many keystrokes have been made, the probability that a selected symbol was not the target symbol is simply the probability that the last keystroke was in error, i.e., $1-p$.

Finally, if backarrow is selected, the previous position is revisited, and the probabilities are reset as though no prior selection had been made.

4 Empirical results

4.1 Subjects and scan rate calibration

We recruited 10 native English speakers between the ages of 24 and 48 years, who had not used our typing interface, are not users of scanning interfaces for typing, and have typical motor function. Each subject participated in two sessions, one for training and calibration of scan rates; and another for testing. We use the phrase set from MacKenzie and Soukoreff (2003) to evaluate typing performance. Of the 500 phrases in that set, 20 were randomly set aside for testing, the other 480 available during training and calibration phases. Five of the 20 evaluation

strings were used in this study. We used an Ablenet Jellybean® button as the binary switch. For these trials, to estimate error rates in modeling, we fixed $p = 0.95$, i.e., 5% error rate.

The scan rate for row/column scanning is typically different than for Huffman or linear scanning, since row/column scanning methods allow for anticipation: one can tell from the current highlighting whether the desired row or column will be highlighted next. For the Huffman and linear scanning approaches that we are investigating, that is not the case: any cell can be highlighted (or symbol displayed) at any time, even multiple times in a row. Hence the scan rate for these methods depends more on reaction time than row/column scanning, where anticipation allows for faster rates.

The scan rate also differs between the two row/column scanning approaches (auto scan and step scan), due to the differences in control needed to advance scanning with a button press versus selecting with a button press. We thus ran scan rate calibration under three conditions: row/column step scan; row/column auto scan; and Huffman scanning, using a unigram language model. The Huffman scanning scan rate was then used for all of the Huffman and linear scanning approaches.

Calibration involved two stages for each of the three approaches, and the first stage of all three is completed before running the second stage, thus familiarizing subjects with all interfaces prior to final calibration. The first stage of calibration starts with slow scan rate (1200 ms dwell time), then speeds up the scan rate by reducing dwell time by 200 ms when a target string is successfully typed. Success here means that the string is correctly typed with less than 10% error rate. The subject gets three tries to type a string successfully at a given scan rate, after which they are judged to not be able to complete the task at that rate. In the first stage, this stops the stage for that method and the dwell time is recorded. In the second stage, calibration starts at a dwell time 500 ms higher than where the subject failed in the first stage, and the dwell time decreases by 100 ms increments when target strings are successfully typed. When subjects cannot complete the task at a dwell time, the dwell time then increases at 50 ms increments until they can successfully type a target string.

Table 1 shows the mean (and std) scan rates (dwell time) for each condition. Step scanning generally had a slower scan rate than auto scanning, and Huffman scanning (unsurprisingly) was slowest.

4.2 Testing stage and results

In the testing stage of the protocol, there were six conditions: (1) row/column step scan; (2) row/column auto scan; (3) Huffman scanning with codes derived from the unigram language model; (4) Huffman scanning with codes derived from the 8-gram language model; (5) Linear scanning on the 6×6 spelling grid with codes derived from the 8-gram language model; and (6) RSVP single letter presentation with codes derived from the 8-gram language model. The ordering of the conditions for each subject was randomized. In each condition, instructions were given (identical to instructions during calibration phase), and the subjects typed practice phrases until they successfully reached error rate criterion performance (10% error rate or lower), at which point they were given the test phrases to type.

Recall that the task is to type the stimulus phrase exactly as presented, hence the task is not complete until the phrase has been correctly typed. To avoid non-termination scenarios – e.g., the subject does not recognize that an error has occurred, what the error is, or simply cannot recover from cascading errors – the trial is stopped if the total errors in typing the target phrase reach 20, and the subject is presented with the same target phrase to type again from the beginning, i.e., the example is reset. Only 2 subjects in the experiment had a phrase reset in this way (just one phrase each), both in row/column scanning conditions. Of course, the time and keystrokes spent typing prior to reset are included in the statistics of the condition.

Table 1 shows the mean (and std) of several measures for the 10 subjects. Speed is reported in characters per minute. Bits per character represents the number of keypress and non-keypress (timeout) events that were used to type the symbol. Note that bits per character does not correlate perfectly with speed, since a non-keypress bit due to a timeout takes the full dwell time, while the time for a keypress event may be less than that full time. For any given symbol the bits may involve making an error, followed by deleting the erroneous symbol and retyping the correct symbol. Alternately, the subject may scan pass the target symbol, but still return to type it correctly, resulting in extra keystrokes, i.e., a longer binary code than optimal. In addition to the mean and standard deviation of bits per character, we present the optimal could be achieved with each method. Finally we characterize the errors that are made by subjects by the error rate, which is the num-

Scanning condition	Scan rate (ms) mean (std)	Speed (cpm) mean (std)	Bits per character mean (std)	opt.	Error rate mean (std)	Long code rate mean (std)
row/column step scan	425 (116)	20.7 (3.6)	8.5 (2.6)	4.5	6.3 (5.1)	29.9 (19.0)
auto scan	310 (70)	19.1 (2.2)	8.4 (1.2)	4.5	5.4 (2.8)	33.8 (11.5)
Huffman unigram	475 (68)	12.5 (2.3)	8.4 (1.9)	4.4	4.4 (2.2)	39.2 (13.5)
8-gram	475 (68)	23.4 (3.7)	4.3 (1.1)	2.6	4.1 (2.2)	19.3 (14.2)
Linear grid 8-gram	475 (68)	23.2 (2.1)	4.2 (0.7)	3.4	2.4 (1.5)	5.0 (4.1)
RSVP 8-gram	475 (68)	20.3 (5.1)	6.1 (2.6)	3.4	7.7 (5.4)	5.2 (4.0)

Table 1: Typing results for 10 users on 5 test strings (total 31 words, 145 characters) under six conditions.

ber of incorrect symbols typed divided by the total symbols typed. The long code rate is the percentage of correctly typed symbols for which a longer than optimal code was used to type the symbol, by making an erroneous selection that does not result in typing the wrong symbol.

We also included a short survey, using a Likert scale for responses, and mean scores are shown in Table 2 for four questions: 1) I was fatigued by the end of the trial; 2) I was stressed by the end of the trial; 3) I liked this trial; and 4) I was frustrated by this trial. The responses showed a consistent preference for Huffman and linear grid conditions with an 8-gram language model over the other conditions.

Survey Question	Row/Column step	auto	Huffman 1-grm	8-grm	Linear grid	RSVP
Fatigued	3.2	2.4	3.6	2.0	2.4	2.8
Stressed	2.7	2.4	2.7	1.5	1.8	2.6
Liked it	2.2	3.3	2.3	4.2	3.8	3.2
Frustrated	3.2	1.7	3.1	1.7	1.7	2.3

Table 2: Mean Likert scores to survey questions (5 = strongly agree; 1 = strongly disagree)

4.3 Discussion of results

While this is a preliminary study of just 10 subjects, several things stand out from the results. First, comparing the three methods using just unigram frequencies to inform scanning (row/column and Huffman unigram), we can see that Huffman unigram scanning is significantly slower than the other two, mainly due to a slower scan rate with no real improvement in bits per character (real or optimal). All three methods have a high rate of longer than optimal codes, leading to nearly double the bits per character that would optimally be required.

Next, with the use of the 8-gram language model in Huffman scanning, both the optimal bits per character and the difference between real and optimal are reduced, leading to nearly double the speed. Interestingly, use of the linear code on the grid leads to fewer bits per character than Huffman scanning, despite nearly 1 bit increase in optimal bits per charac-

ter, due to a decrease in error rate and a very large decrease in long code rate. We speculate that this is because highlighting a single cell at a time draws the eye to that cell, making visual scanning easier.

Finally, despite using the same model, RSVP is found to be slightly slower than the Huffman 8-gram or Linear grid conditions, though commensurate with the row/column scanning, mainly due to an increase in error rate. Monitoring a single cell, recognizing symbol identity and pressing the switch is apparently somewhat harder than finding the symbol on a grid and waiting for the cell to light up.

5 Summary and future directions

We have presented methods for including language modeling in simple scanning interfaces for typing, and evaluated performance of novice subjects with typical motor control. We found that language modeling can make a very large difference in the usability of the Huffman scanning condition. We also found that, despite losing bits to optimal Huffman coding, linear coding leads to commensurate typing speed versus Huffman coding presumably due to lower cognitive overhead of scanning and thus fewer mistakes. Finally, we found that RSVP was somewhat slower than grid scanning with the same language model and code.

This research is part of a program to make the simplest scanning approaches as efficient as possible, so as to facilitate the use of binary switches for individuals with the most severe impairments, including ERP for locked-in subjects. While our subjects in this study have shown slightly better performance using a grid versus RSVP, these individuals have no problem with visual scanning or fixation on relatively small cells in the grid. It is encouraging that subjects can achieve nearly the same performance with an interface that simply displays an option and requests a yes or a no. We intend to run this study with subjects with impairment, and are incorporating the interfaces with an ERP detection system for use as a brain-computer interface.

Acknowledgments

References

D. Anson, P. Moist, M. Przywars, H. Wells, H. Saylor, and H. Maxime. 2004. The effects of word completion and word prediction on typing rates using on-screen keyboards. *Assistive Technology*, 18(2):146–154.

G. Baletsa, R. Foulds, and W. Crochetiere. 1976. Design parameters of an intelligent communication device. In *Proceedings of the 29th Annual Conference on Engineering in Medicine and Biology*, page 371.

M. Baljko and A. Tam. 2006. Indirect text entry using one or two keys. In *Proceedings of the Eigth International ACM Conference on Assistive Technologies (ASSETS)*, pages 18–25.

B. Carpenter. 2005. Scaling high-order character language models to gigabytes. In *Proceedings of the ACL Workshop on Software*, pages 86–99.

J.J. Darragh, I.H. Witten, and M.L. James. 1990. The reactive keyboard: A predictive typing aid. *Computer*, 23(11):41–49.

L.A. Farwell and E. Donchin. 1988. Talking off the top of your head: toward a mental prosthesis utilizing event-related brain potentials. *Electroenceph Clin. Neurophysiol.*, 70:510–523.

J.P. Hansen, A.S. Johansen, D.W. Hansen, K. Itoh, and S. Mashino. 2003. Language technology in a predictive, restricted on-screen keyboard with ambiguous layout for severely disabled people. In *Proceedings of EACL Workshop on Language Modeling for Text Entry Methods*.

D.A. Huffman. 1952. A method for the construction of minimum redundancy codes. In *Proceedings of the IRE*, volume 40(9), pages 1098–1101.

R. Kneser and H. Ney. 1995. Improved backing-off for m-gram language modeling. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 181–184.

G.W. Lesher, B.J. Moulton, and D.J. Higginbotham. 1998. Techniques for augmenting scanning communication. *Augmentative and Alternative Communication*, 14:81–101.

J. Li and G. Hirst. 2005. Semantic knowledge in word completion. In *Proceedings of the 7th International ACM Conference on Computers and Accessibility*.

I.S. MacKenzie and R.W. Soukoreff. 2003. Phrase sets for evaluating text entry techniques. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*, pages 754–755.

J. Perelmouter and N. Birbaumer. 2000. A binary spelling interface with random errors. *IEEE Transactions on Rehabilitation Engineering*, 8(2):227–232.

B. Roark. 2009. Open vocabulary language modeling for binary response typing interfaces. Technical Report #CSLU-09-001, Center for Spoken Language Processing, Oregon Health & Science University. cslu.ogi.edu/publications/ps/roark09.pdf.

I. Schadle. 2004. Sibyl: AAC system using NLP techniques. In *Proceedings of the 9th International Conference on Computers Helping People with Special needs (ICCHP)*, pages 1109–1015.

E.W. Sellers and E. Donchin. 2006. A p300-based brain-computer interface: initial tests by als patients. *Clinical Neuropsysiology*, 117:538–548.

E.W. Sellers, G. Schalk, and E. Donchin. 2003. The p300 as a typing tool: tests of brain-computer interface with an als patient. *Psychophysiology*, 40:77.

C.E. Shannon. 1950. Prediction and entropy of printed English. *Bell System Technical Journal*, 30:50–64.

K. Trnka, D. Yarrington, K.F. McCoy, and C. Pennington. 2006. Topic modeling in fringe word prediction for AAC. In *Proceedings of the International Conference on Intelligent User Interfaces*, pages 276–278.

K. Trnka, D. Yarrington, J. McCaw, K.F. McCoy, and C. Pennington. 2007. The effects of word prediction on communication rate for AAC. In *Proceedings of HLT-NAACL; Companion Volume, Short Papers*, pages 173–176.

H. Trost, J. Matiasek, and M. Baroni. 2005. The language component of the FASTY text prediction system. *Applied Artificial Intelligence*, 19(8):743–781.

T. Wandmacher and J.Y. Antoine. 2007. Methods to integrate a language model with semantic information for a word prediction component. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 506–513.

T. Wandmacher, J.Y. Antoine, F. Poirier, and J.P. Departe. 2008. Sibylle, an assistive communication system adapting to the context and its user. *ACM Transactions on Accessible Computing (TACCESS)*, 1(1):6:1–30.

D.J. Ward, A.F. Blackwell, and D.J.C. MacKay. 2002. DASHER – a data entry interface using continuous gestures and language models. *Human-Computer Interaction*, 17(2-3):199–228.

I.H. Witten and T.C. Bell. 1991. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094.