The Stata Journal

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press. Stata and Mata are registered trademarks of StataCorp LP.

The Stata Journal (2006)
6, Number 4, pp. 435–460

Sequence analysis with Stata

Christian Brzinsky-Fay
Wissenschaftszentrum Berlin
Berlin, Germany
brzinsky-fay@wz-berlin.de

Ulrich Kohler
Wissenschaftszentrum Berlin
Berlin, Germany
kohler@wz-berlin.de

Magdalena Luniak
Wissenschaftszentrum Berlin
Berlin, Germany
luniak@wz-berlin.de

Abstract. We describe a general strategy to analyze sequence data and introduce SQ-Ados, a bundle of Stata programs implementing the proposed strategy. The programs include several tools for describing and visualizing sequences as well as a Mata library to perform optimal matching using the Needleman–Wunsch algorithm. With these programs Stata becomes the first statistical package to offer a complete set of tools for sequence analysis.

Keywords: st0111, sqclusterdat, sqclustermat, sqdes, sqindexplot, sqom, sqparcoord, sqset, sqstatlist, sqstatsum, sqstattab1, sqstattab2, sqstattabsum, sqtab, sequence analysis, optimal matching, cluster analysis, panel data, longitudinal data, explorative data analysis, sequence index plot

1 Introduction

Sequence data arise in many scientific fields, such as biology, where DNA sequences constitute the basic foundation of life, and the social sciences, where researchers investigate life courses, marital histories, and employment profiles. A sequence is defined as an ordered list of elements, where an element can be a certain status (e.g., employment or marital status), a physical object (e.g., base pair of DNA, protein, or enzyme), or an event (e.g., a dance step or bird call). The positions of the elements are fixed and ordered by elapsed time or by another more or less natural order (see figure 1).

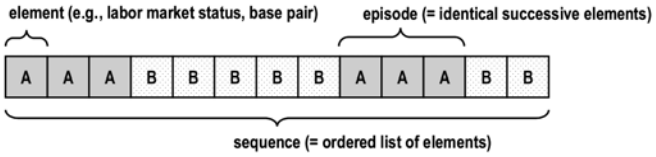

Figure 1: Sample sequence

Sequence data share some of the properties of cross-sectional time-series and survival data. However, unlike the former, the positions in a sequence refer to a relative, not an absolute, time point. Moreover, sequences are generally seen as an entity of their own,

and the interest is in the sequential character of all elements together. Unlike survival data, they do not involve a hazard or censoring.

Including the sequential information in the research design increases the complexity of the analysis because the number of possible sequences grows exponentially with the sequence length. For example, with three elements and a sequence length of 36, one can form $3^{36} = 1.5\times10^{17}$ sequences. Dealing with sequence data therefore raises two questions: how can the sequential character of the data be maintained without reducing it to single events, and how can the variation in the sequences be optimally reduced. The strategy proposed here to analyze sequence data involves five steps:

1. *description*, i.e., tabulation of sequences and calculation of indicators for the characteristics of each sequence;
2. *visualization* with sequence index plots or parallel-coordinates plots;
3. *comparison* using distance measures obtained via optimal matching (OM);
4. *grouping* of "similar sequences", based on the results of the comparison step using techniques like cluster analysis or multidimensional scaling; and
5. *application* by using the grouped sequences as dependent or independent variables in standard regression models or other confirmatory analyses.

Here we will describe steps 1–4 in more detail. Step 5 is omitted because it involves only standard statistical techniques such as cross-tabulation and regression models. As we proceed, we will introduce a bundle of user-written Stata commands for sequence analysis ("SQ-Ados") that make Stata the first general statistical package to offer tools for all steps of sequence analysis.[1] Because of space constraints, we cannot describe each command in detail here. We refer readers to the help files of the respective commands for a full description of the options. An introduction to the commands can also be found in `help sq`, and a software demonstration is provided in `help sqdemo`.

2 Preliminaries

We will use artificial data on the employment status of 500 graduates over a period of up to 36 months after leaving high school. The data resemble a cross-sectional time-series dataset like the British Household Panel Study, http://www.iser.essex.ac.uk/ulsc/bhps/, or the German Socio-Economic Panel, http://www.diw.de/english/sop/. The listing below shows the *positions* 1–10 of the respondent with `id` 43. The sequence starts with the

1. The freeware program TDA (http://www.stat.ruhr-uni-bochum.de/tda.html) performs OM and calculates many descriptive statistics, but it lacks easily accessible visualization features and cluster analysis tools. SAS produces sequence index plots with the help of a user-written program (Scherer 2001) but does not perform OM itself. "Optimize" by Andrew Abbott (http://home.uchicago.edu/~aabbott/om.html) performs OM and several graphical displays for some sequences. Finally, a variety of specialized biological sequencing programs for OM exist that tend to be optimized for small numbers of long sequences.

element 3 (i.e., employment), changes to vocational education, and ends with inactivity at the 10th position.

```
. use youthemp, clear
(youth sequences: 36 months, wide)
. list st1-st10 if id == 43, nolabel
```

	st1	st2	st3	st4	st5	st6	st7	st8	st9	st10
43.	3	3	3	2	2	2	2	2	5	5

```
. label list lbstat
lbstat:
           1 higher education
           2 vocational education
           3 employment
           4 unemployment
           5 inactivity
```

The example sequence data are in wide form, but before the commands for sequence analysis can be used, the data need to be in long form and must be `sqset`. The former can easily be accomplished with the official Stata command [D] **reshape**:

```
. reshape long st, i(id) j(order)
  (output omitted)
```

The `sqset` command serves a similar function to `tsset` (see [TS] **tsset**) and `stset` (see [ST] **stset**) for cross-section time-series and survival time analysis, respectively. It is used to declare the variable that holds the elements of a sequence, the variable that holds the order of the elements, and the variable that uniquely identifies each sequence. Its syntax diagram is

`sqset` *elementvar idvar ordervar* [`, clear` [`ltrim` | `rtrim` | `trim`] `keeplongest`]

where *elementvar* is the name of the variable that contains the elements, *idvar* is the sequence identifier, and *ordervar* is the variable that defines the order of a sequence. The standard use of `sqset` requires no options, which is sufficient for our example here. The available options can be used if there are incomplete sequences; `sqset` checks for the type of incompleteness and suggests the proper option. Often sequences derived from unbalanced panels contain missing values at the beginning or the end of the sequence. The option `ltrim` cuts off the missing values at the beginning by aligning all sequences to the first position. The option `rtrim` deletes the missing values at the end of a sequence, and `trim` does both. Finally, the option `keeplongest` is used to keep only the longest contiguous part of each sequence. Read `help sqset` carefully and the respective section in `help sq` before using `keeplongest`.

```
. sqset st id order
        element variable:  st, 1 to 5
        identifier variable:  id, 1 to 500
        order variable:  order, 1 to 36
```

3 Description

Some general descriptive properties of sequences should be considered to reduce the enormous amount of information contained in even a few relatively short sequences. Such descriptions can be obtained in three ways:

- The `sqtab` command produces frequency tables of sequences.
- The `sqdes` command assesses the concentration of the sequences.
- A set of `egen` functions generates variables containing descriptive information about each sequence, and a further set of commands describes these e-generated variables.

3.1 Frequency tables of sequences

The `sqtab` command displays frequency tables of sequences. Here is the syntax diagram:

`sqtab` [*varname*] [*if*] [*in*] [`, ranks(`*numlist*`) se so nosort gapinclude` *tabulate_options*]

Without further options, `sqtab` produces a frequency table of all sequences in the dataset, showing as much of the sequence as possible.

```
. sqtab
```

Sequence-Pattern	Freq.	Percent	Cum.
44444444444444444444444444444444444	38	7.60	7.60
33333333333333333333333333333333333	28	5.60	13.20
55555555555555555555555555555555555	18	3.60	16.80
44444444444411111111111111111111111	10	2.00	18.80
55555555555511111111111111111111111	9	1.80	20.60
55511111111111111111111111111111111	7	1.40	22.00
44444444444444444444444555555555555	5	1.00	23.00
55111111111111111111111111111111111	5	1.00	24.00
22222222222222222222222222222222222	4	0.80	24.80
44433333333333333333333333333333333	4	0.80	25.60
44444444444433333333333333333333333	4	0.80	26.40
(*output omitted*)			
55555555555555555555555555553433333	1	0.20	99.80
55555555555555555555555555555444444	1	0.20	100.00
Total	500	100.00	

The table shows that the sequence containing only elements designated 4 (unemployment) is the most frequent sequence in the dataset, followed by a sequence containing only employment. There are many sequences that are observed only once, which we call *unique* sequences.

In general, `sqtab` with no options produces large tables because there are often many unique sequences in the data. Therefore, the default order of the resulting table is such

that the most frequent sequences are at the top. This feature can be turned off by the `nosort` option. Also it is possible to restrict the output to the most frequent sequences by using the `ranks()` option. The following example uses `ranks(1/10)` to show only the 10 most frequent sequences.[2]

```
. sqtab, ranks(1/10)
```

Sequence-Pattern	Freq.	Percent	Cum.
44444444444444444444444444444444444	38	28.79	28.79
33333333333333333333333333333333333	28	21.21	50.00
55555555555555555555555555555555555	18	13.64	63.64
44444444444111111111111111111111111	10	7.58	71.21
55555555555111111111111111111111111	9	6.82	78.03
55511111111111111111111111111111111	7	5.30	83.33
44444444444444444444444444555555555	5	3.79	87.12
55111111111111111111111111111111111	5	3.79	90.91
22222222222222222222222222222222222	4	3.03	93.94
44433333333333333333333333333333333	4	3.03	96.97
44444444444433333333333333333333333	4	3.03	100.00
Total	132	100.00	

Further simplified, the `sqtab` command allows two straightforward definitions of sequence similarity. The `so` option treats identically all sequences that have the *same order* of elements; i.e., the sequence A-B-B-A would be treated the same as A-B-A-A because the elements A and B appear in the same order in both sequences (first A, then B, and then A again). The `se` option considers sequences identical if they consist of the *same elements*, such as the sequences B-A-A-B and A-B-B-A, because both sequences consist of the elements A and B only. Here are examples of both options:[3]

```
. sqtab, ranks(1/10) so
```

Sequence-Order	Freq.	Percent	Cum.
4	38	16.45	16.45
3	28	12.12	28.57
43	28	12.12	40.69
51	28	12.12	52.81
5	18	7.79	60.61
41	17	7.36	67.97
4343	13	5.63	73.59
45	11	4.76	78.35
31	10	4.33	82.68
313	10	4.33	87.01
32	10	4.33	91.34
323	10	4.33	95.67
53	10	4.33	100.00
Total	231	100.00	

2. If subsequent sequence patterns have the same frequency as the higher number specified in `ranks()`, they are also displayed.

3. An application of the two similarity definitions from political sociology can be found in Kohler (2002).

```
. sqtab, ranks(1/10) se
```

Sequence-Elements	Freq.	Percent	Cum.
34	68	17.85	17.85
134	42	11.02	28.87
345	38	9.97	38.85
4	38	9.97	48.82
13	36	9.45	58.27
15	31	8.14	66.40
23	29	7.61	74.02
3	28	7.35	81.36
45	27	7.09	88.45
135	22	5.77	94.23
14	22	5.77	100.00
Total	381	100.00	

The `gapinclude` option is used to include sequences with gaps in the tabulation. By default, all SQ-Ados exclude sequences with gaps, although this is required only for the OM algorithm and otherwise can be overridden with the `gapinclude` option. Other ways of dealing with broken sequences are described in `help sq`.

Finally, if an optional variable name is specified, a cross tabulation of the sequences with the specified variable will be displayed.

3.2 Concentration of sequences

Tabulating sequences is connected to the concept of concentration. In the tabulation on page 439, the most frequent sequence is shared by 38 of the 500 respondents. In the limiting cases when all (no) respondents share the same sequence, there is a high (low) concentration of sequences. Hence, the concentration is lower when there are more unique sequences.

The `sqdes` command provides information about the concentration or diversification of sequences. Its syntax diagram is

`sqdes` [*if*] [*in*] [`, so se graph gapinclude`]

where `so` and `se` refer to the similarity concepts described in section 3.1. The `graph` option is used to display a graphical representation of the output.

```
. sqdes
   # of observed sequences: 500
overall # of obs. elements: 5
       max sequence length: 36
 # of producible sequences: 1.455e+25
```

Observations	Sequences	% of observed	Cum.
1	309	61.8	61.8
2	22	4.4	66.2
3	5	1	67.2
4	3	.6	67.8
5	2	.4	68.2
7	1	.2	68.4
9	1	.2	68.6
10	1	.2	68.8
18	1	.2	69
28	1	.2	69.2
38	1	.2	69.4
Total	347	69.4	

In its header, the output of `sqdes` shows that we have observed 500 sequences. Among these 500 sequences, we have observed five different elements and up to 36 positions, implying $5^{36} = 1.455 \times 10^{25}$ theoretically producible sequences.

Among the 500 observed sequences there are only 347 *different* sequences. This number is shown in the last row of the table in the output of `sqdes`. In the limiting case when all observed sequences were unique (no concentration), the division of the number of different sequences by the number of observed sequences would be 1, whereby this number would converge to zero when all observed sequences were equal (high concentration). Here this measure of concentration is 69.4%.

The remaining numbers shown in the table are a breakdown of these overall concentration measures. Three hundred nine of the 347 observed sequences are unique (61.8% of the 500 observed sequences); 22 further sequences (4.4%) are shared by two persons, etc.

3.3 Sequence-specific descriptions

The SQ-Ados provide ways of describing important characteristics of observed sequences. Examples of such characteristics include the length of the sequences, the number of elements in each sequence, and the number of status changes in the sequence. Several such quantities can be stored as a variable in the dataset by using a bundle of `egen` functions (see `help sqegen`). A second bundle of commands are used to describe these e-generated variables (see `help sqstat`).

The syntax of the `egen` functions follows the standard syntax; see [D] **egen**. The following functions are available:

`sqelemcount()` [`, element(#) gapinclude`] generates a variable holding the number of different elements in each sequence. The `gapinclude` option generates these counts for sequences with gaps, treating gaps as just another element.

`sqepicount()` [`, element(#) gapinclude`] separates a sequence into sections of equal elements (called *episodes*) and generates a variable holding the number of episodes for each sequence. The number of episode changes can be calculated by subtracting one from the number of episodes. The `element()` option can be used to restrict the episode count to the specified elements.

`sqlength()` [`, element(#) gapinclude`] generates a variable holding the length of each observed sequence, and the `element()` option can be used as above.

`sqgapcount()` generates a variable holding the number of gap episodes in each sequence.

`sqgaplength()` generates a variable holding the overall length of gap episodes in each sequence.

Examples of the `egen` functions are given below. Unlike common uses of `egen`, nothing is required inside the parentheses because the functions use the declarations provided with `sqset`:

```
. egen length = sqlength()
. egen length1 = sqlength(), element(1)
. egen elemnum = sqelemcount()
. egen epinum = sqepicount()
```

When using the variables generated with the `sq-egen` functions, keep in mind that the data are in long form, whereas the new variables refer to the sequences as entities. To further describe the variables, one can reshape the data back to wide format. However, a more convenient way to describe the new variables is provided by the `sqstat` commands, which summarize, tabulate, and list the variables generated with the `sq-egen` functions as if the data were in wide format, and the names of the variables generated by `sq-egen` are automatically processed.

The following `sqstat` commands are available:

`sqstatlist` [*varlist*] [*if*] [*in*] [`, ranks(`*numlist*`) replace` *list_options*], if given without *varlist*, lists all variables generated by the `sq-egen` bundle. With a *varlist* only the specified variables are listed. The option `ranks(`*numlist*`)` restricts the listing to the most frequent sequences, whereas `replace` keeps the listed data as a dataset in memory.

`sqstatsum` [*varlist*] [*if*] [*in*] [`,` *summarize_options*], if given without *varlist*, summarizes all variables generated by the `sq-egen` bundle. With *varlist*, only the specified variables are summarized.

`sqstattab1` [*varlist*] [*if*] [*in*] [`,` *tab1_options*], if given without *varlist*, produces one-way frequency tables of all variables generated by the `sq-egen` bundle. With *varlist*, only the specified variables are used.

`sqstattab2` *varname*$_1$ [*varname*$_2$] [*if*] [*in*] [`,` *tab2_options*], if given without *varname*$_2$, displays a two-way table of *varname*$_1$ against all variables generated by the `sq-egen` bundle. If specified with *varname*$_2$, a two-way table of the two specified variables is displayed.

`sqstattabsum` *varname*$_1$ [*varname*$_2$] [*if*] [*in*] [`, format(%`*fmt*`)` *tabsum_options*] summarizes all e-generated variables for categories of the specified variables.

Here are some examples, which assume that the above `egen` commands have been executed. A further description of the `sqstat` commands can be found in `help sqstat`:

```
. sqstatsum
```

Variable	Obs	Mean	Std. Dev.	Min	Max
length	500	36	0	36	36
length1	500	7.44	11.14481	0	35
elemnum	500	2.244	.8329776	1	5
epinum	500	3.122	1.843343	1	12

We have 500 sequences, each with a length of 36, which also implies that the mean is 36. Some of the sequences contain the element 1 (education), and there is at least one sequence where 35 positions contain this element. On average only seven positions are occupied by education, however. The number of elements in all sequences is at least 1, and there are some sequences that contain all five possible elements. The maximum number of episodes is even higher, implying that some sequences oscillate between elements.

```
. sqstattab1 elemnum
-> tabulation of elemnum
```

Number of different elements in sequence	Freq.	Percent	Cum.
1	88	17.60	17.60
2	238	47.60	65.20
3	141	28.20	93.40
4	30	6.00	99.40
5	3	0.60	100.00
Total	500	100.00	

Only three sequences consist of all five elements, and 238 of the 500 sequences consist of only two elements, implying that some elements typically do not appear together, although more analysis is necessary to verify this claim.

(*Continued on next page*)

```
. sqstattabsum sex
```

sex	Summary of Length of sequence Mean	Std. Dev.	Freq.
male	36	0	256
female	36	0	244
Total	36	0	500

sex	Summary of Length of episodes of element 1 Mean	Std. Dev.	Freq.
male	7	11	256
female	7	11	244
Total	7	11	500

sex	Summary of Number of different elements in sequence Mean	Std. Dev.	Freq.
male	2	1	256
female	2	1	244
Total	2	1	500

sex	Summary of Number of episodes Mean	Std. Dev.	Freq.
male	3	2	256
female	3	2	244
Total	3	2	500

Here the output reveals that the sequences do not vary with sex.

4 Visualization

A graphical representation is advisable in the usual case where the sequences are complex. An often-used technique to visualize sequence data is the so-called sequence index plot (Scherer 2001; Kogan 2003; Brüderl and Scherer 2004; Brzinsky-Fay 2006). Parallel-coordinates plots are an alternative that is used less often (Kohler 2002). The SQ-Ados contain commands for both kinds of plots.

4.1 Sequence index plots

Sequence index plots were proposed by Scherer (2001). The idea is to draw a horizontal line for each sequence, separating the elements with different colors. As shown in a previous publication (Kohler and Brzinsky-Fay 2005), such plots can be easily produced with `graph twoway hbar` or `graph twoway hline`. The command `sqindexplot` implements and further extends the idea by using the following syntax:

`sqindexplot` [*if*] [*in*] [`, ranks(`*numlist*`) se so order(`*varname*`) by(`*varname*`) color(`*colorstyle*`) gapinclude` *twoway_options*]

A simple application of

```
. sqindexplot
```

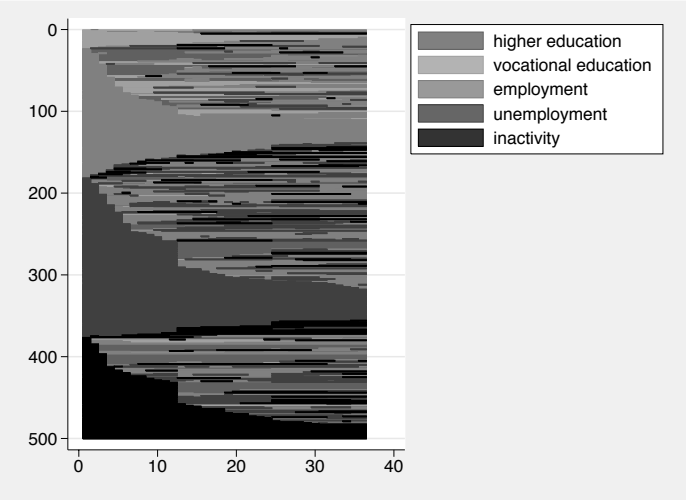

Figure 2: Sequence index plot

produces the graph shown in figure 2, which is far from optimal. To fine-tune the graph, `sqindexplot` allows all options that are available for `graph twoway`. When applying these options, consider several points:

- In general, color versions of sequence index plots are more sensible than black-and-white versions. The `color()` option allows fine-tuning of the colors used for the elements.
- With many observations, there is a tendency to overplot the lines, which has the effect of overrepresenting elements with higher category values (levels). The effect depends on the printer and/or screen resolution and can be adjusted by tuning the aspect ratio (see Cox 2004). It might also be sensible to restrict the graph to the most frequent sequences with the `ranks()` option or to plot groups of sequences separately with standard graphic options, e.g., `by(`*varname*`, yrescale)`.
- Sequence index plots depend heavily on the order of the sequences along the vertical axis. Without further options, a naive algorithm is used to order the sequences; however, the `order()` option sorts the sequences according to a user-defined variable. It is sensible to use the results of the comparison step (section 5) or the grouping step (section 6) to order the sequences in a sequence index plot.

Besides the standard form, `sqindexplot` produces similar plots after applying the "same order" or "same elements" similarity (section 3.1). The respective options are `so` and `se`.

4.2 Parallel-coordinates plots

Parallel-coordinates plots can easily be produced by Stata's official `graph twoway line` command and are also implemented as statistical graphs (see [XT] **xtline**). In their standard form, these plots are especially helpful for cross-sectional time-series data of continuous variables (see the examples in Diggle, Liang, and Zeger [1994, 12]). With sequence data, the variables used for the vertical axis are usually categorical (the elements), and there is normally no relationship between the position in the sequence and the element at that position. Consequently, standard parallel-coordinates plots become unreadable for even moderate numbers of sequences, which is why they are seldom used as a graphical device for sequence data outside teaching. However, they can reveal valuable information.

To our knowledge, the only application of parallel-coordinates plots for many sequences is in Kohler (2002), which `sqparcoord` builds on. The basic idea of the program is to add optical effects to highlight frequent sequences and distinguish the lines from different sequences. Its syntax is

`sqparcoord` [*if*] [*in*] [`, ranks(`*numlist*`) so offset(`#`) wlines(`#`)`
`gapinclude` *twoway_options*]

where `ranks(`*numlist*`)` and `so` have the same meaning as for the sequence index plot.[4]

Figure 3 shows an example using the `wlines()` option and plots the elements of sequences along the vertical axis and the position (e.g., time points) along the horizontal axis. The sequences are drawn with a line that connects the elements in position order. The `wlines()` option draws thicker lines for frequent sequences, where the number in parentheses controls the weighting factor. Generally, the thicker the line, the more frequent the sequence. In our example, the graph shows that the "only unemployment" sequence is the most frequent sequence, followed by "only employment" and "only inactivity". The dense (dark) region of the graph shows that changes between employment and unemployment are frequent over the whole period, whereas changes between education and other elements decrease slightly over time.

4. "Same elements" similarity is not applicable for parallel-coordinates plots.

```
. sqparcoord, ylabel(1(1)5, valuelabel angle(0)) wlines(2)
```

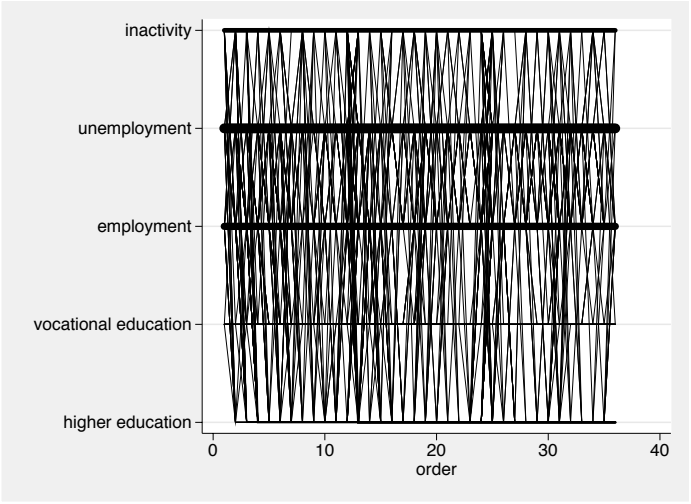

Figure 3: Parallel-coordinates plot with `wlines()`

Figure 4 shows an example of the `offset()` option, combined with `so`, `ranks()`, and `wlines()`. The figure was drawn with the following command:

```
. sqparcoord, so ranks(1/10) offset(.5) wlines(1)
> ylabel(1(1)5, valuelabel angle(0))
```

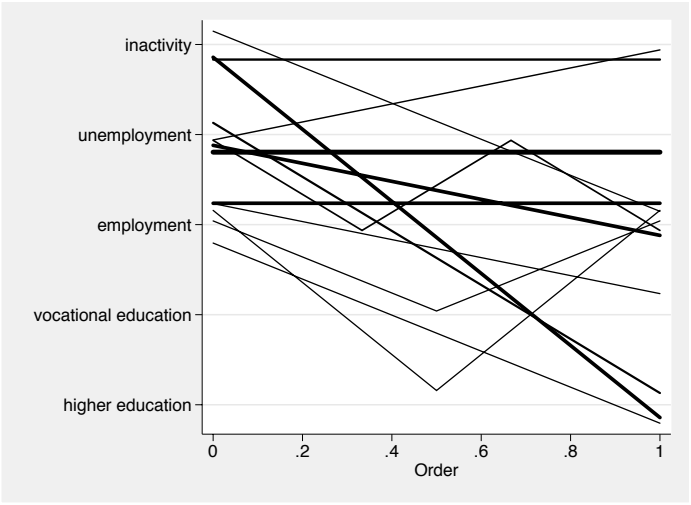

Figure 4: Parallel-coordinates plot with `offset()`, `ranks()`, and `wlines()`

With `offset()`, the lines for each sequence are slightly displaced along the vertical axis, where the number in parentheses controls the amount of displacement. This arrangement makes following the path of the individual sequences easier, which is especially useful for plots with just a few sequences. The plot reveals, among other things, that many of the most frequent sequences end up in employment or higher education.

5 Comparison

In the comparison step, one has to determine how sequences should be compared and how the difference between two sequences should be measured. For this step, the so-called Levenshtein distance is used, a measure from information technology that basically counts the number of operations needed to transform one string into another (Levenshtein 1966). It has been applied to various fields such as plagiarism detection, analysis of DNA sequences (Needleman and Wunsch 1970), ritual dances (Abbott and Forrest 1986), and the succession of lynchings in southern states of the United States (Stovel 2001). We informally introduce the idea of the Levenshtein distance and then go on to explain the functionality of our Stata implementation. A formal description of the computations necessary to derive the Levenshtein distance is given in section 7.1. More information can be found in Sankoff and Kruskal (1983), Waterman (1995), and Rohwer and Pötter (2005, sec. 6.7.2).

5.1 OM

Assume that we observe the following sequences of length 12 for two people, where each element refers to an employment status in a specific month after an arbitrary starting point:

Individual 1	ed	ed	ed	em	em	ue	ue	em	em	em	em	em
Individual 2	ed	ed	em	em	em	em	ue	ue	ue	ue	em	em

Individual 1 spent 3 months in education, after which he or she was employed for 2 months, then unemployed for another 2 months, and finally landed a job for the last 5 months. Individual 2 was in education for 2 months, then employed for 4 months, followed by an unemployment period of 4 months, and then employed again. A simple measure for the distance between these two sequences can be constructed by aligning both sequences and using a penalty, s, whenever the elements at a specific position differ:

ed	ed	ed	em	em	ue	ue	em	em	em	em	em
ed	ed	em	em	em	em	ue	ue	ue	ue	em	em
0	0	s	0	0	s	0	s	s	s	0	0

For an overall distance measure, one could multiply s by the number of differences. Such a measure largely follows the idea of traditional distance measures, such as the

Euclidean distance, but is *not* used for sequence analysis. The idea of the penalty is, however, important, and we will speak of substitution costs for this penalty. One could use different substitution costs for different combinations of discrepant elements.

The reason why this distance measure is not used in sequence analysis can be illustrated using two example sequences. From a visual inspection, one gets the impression that there is some similarity between the sequences that this distance measure neglects. Both sequences show the same succession of episodes; i.e., they begin with education, then have a short employment episode, followed by a short unemployment episode, and finally are employed again. The sequences differ only in episode duration.

To cope with this kind of order similarity, consider the following alignment of the two sequences:

ed	ed	ed	em	em			ue	ue			em	em	em	em	em
ed	ed		em	em	em	em	ue	ue	ue	ue	em	em			
0	0	d	0	0	d	d	0	0	d	d	0	0	d	d	d

Here we have shifted some episodes of both sequences to the right; i.e., we have inserted gaps. Below the alignment we have used 0 if the aligned elements are equal and d if we have inserted a gap. The distance between the two sequences depends on the number of insertions and on the value of d, which we will refer to as the *indel-cost*.[5]

Now consider the following example, where we have not changed the sequences but aligned them differently by shifting the end of the second sequence to the left:

ed	ed	ed	em	em		ue	ue			em	em	em	em	em
ed	ed	em	em	em	em	ue	ue	ue	ue	em	em			
0	0	s	0	0	d	0	0	d	d	0	0	d	d	d

Here we have not inserted further gaps in our alignment. Instead, we have accepted that the elements at the third position are different, which is reflected by the substitution cost on the bottom line. We can calculate the overall distance by summing the terms on the bottom line. The overall distance increases with the number of substitutions and insertions and with the respective substitution and indel costs.

So far, the distance measure between two sequences is straightforward. We simply align two sequences in some way, count the number of substitutions and indels, weigh them with the respective costs, and add them all up, which heuristically defines the Levenshtein distance.

The problematic aspect of this definition is the alignment of the sequences, because we are free to insert gaps, delete positions, or accept differences. Thus there is more than one possible alignment of the two sequences, raising the question of which alignment to choose. The answer is to choose the alignment with the minimum distance, which

5. *Indel* is a combination of *insertion* and *deletion*. The term is used because one can derive the same distance measure by deleting certain positions from one sequence instead of inserting gaps.

is found via the Needleman–Wunsch algorithm (Needleman and Wunsch 1970) in the SQ-Ados; see section 7.1 for details.

Remember the double role of substitution and indel costs in the application of OM. On the one hand, they are terms in the definition of the distance measure; on the other hand, they play a role in the selection of the optimal alignment. It is therefore necessary to define these costs carefully, keeping two considerations in mind.

The first is that there can be good reason to differentiate substitution costs by element combinations. Some researchers refuse to differentiate substitution costs because of lack of theory (Dijkstra and Taris 1995), but often there are striking reasons for differentiation. In general, substitution costs should decline as elements become more similar. Some have proposed that substitution costs should be differentiated empirically by computing them from the category-to-category transition rates in the sequences (Rohwer and Pötter 2005, sec. 6.7.2.5), meaning that less frequent transitions would be more costly than more frequent ones.

The second consideration is the relation between indel and substitution cost. Each substitution can be seen as a combination of one insertion and one deletion (i.e., an insertion in one sequence, followed by a deletion in the other). It is therefore sensible to set substitution costs to double the indel costs. For varying substitution costs, experience with OM has shown that using indel costs that are more than half the highest substitution costs prevents the algorithm from using indel operations, except to set off the difference in sequence length (Macindoe and Abbott 2004, 349). If the position of an element within a sequence is important, one should define the indel costs to be at least half as much as the highest substitution cost. On the other hand, if only the relative position of episodes is important, then one should allow the algorithm to use indel operations for that purpose. Here it seems appropriate to establish indel costs at around 1/10 the largest substitution cost (Macindoe and Abbott 2004, 349).

Finally, if sequences of different length are used, the distance measure will be heavily influenced by the disparity in sequence length because the potential distance between a short and a long sequence is higher than for those of equal length. To avoid this problem, the distance measures have to be standardized by dividing the calculated value by the length of either the sequence with the longer distance or the longest sequence in the dataset.

5.2 The sqom command

SQ-Ados contain the command `sqom` to perform OM. Its syntax diagram is

```
sqom [ if ] [ in ] [ , indelcost(#) subcost(# | rawdistance | mat_exp | matname)
    name(varname) refseqid(spec) full k(#)
    standard(# | cut | longer | longest | none) ]
```

Summary of options

`indelcost(#)` specifies the cost associated with an insertion or deletion in an alignment. The default is `indelcost(1)`.

`subcost(# | rawdistance | mat_exp | matname)` specifies the cost associated with a substitution in an alignment. The default is two times the value specified as the indel cost. Substitution costs may be specified as a number, as an implied formula, or as a full-substitution matrix. Specifying `subcost(3)` will assign a cost of 3 to any substitution in an alignment, regardless of how similar the substituted values may be. `subcost(rawdistance)` will use the absolute value of the difference between the two substituted values. A full substitution cost matrix can be created either by specifying the name of a matrix containing the substitution cost or by typing a valid matrix. Such a matrix must be a symmetric $p \times p$ matrix, where p is the number of different elements in all sequences. Specifying a full-substitution cost matrix can increase the running time of the program considerably. The `k()` option might be considered for `sqom` when you are using a full-substitution cost matrix.

`name(varname)` is used to specify the name of the variable that stores the distances. If not specified, `_SQdist` will be used and will be overwritten without warning whenever a `sqom` command without option `full` is invoked.

`refseqid(spec)` is used to select the reference sequence against which all sequences in the dataset are tested. An existing value of the sequence identifier must be specified in the parentheses. By default, the most frequent sequence is used.

`full` is used to perform OM for all sequences in the dataset against all others. The results of these comparisons are stored in the distance matrix "SQdist". Specifying the `full` option will increase the running time of the program considerably.

`k(#)` might be used for `sqom` with `full`, which is used to speed up the calculation of the OM algorithm.[6] A positive integer between 1 and the number of positions of the longest sequence can be given in the parentheses. The increase in speed will be higher with small numbers, but using small numbers can cause the algorithm to miss the optimal alignment between some sequences, especially if substitution costs are high relative to indel costs. See section 7.2 for more information on this option.

6. The implementation of the option `k()` is based partly on the source code of TDA, written by Götz Rohwer and Ulrich Pötter. TDA is a powerful program for transitory data analysis. It is programmed in C and distributed as freeware under the terms of the General Public License. It is downloadable from http://www.stat.ruhr-uni-bochum.de/tda.html.

`standard(#|cut|longer|longest|none)` is used to define the standardization of the resulting distances. With `standard(#)` all sequences are cut to the length #. `standard(cut)` automatically cuts all sequences to the length of the shortest sequence in the dataset. `standard(longer)` divides all distances by the length of the longer sequence of the respective alignment. `standard(longest)` divides all distances by the length of the longest sequence in the dataset, which is the default. `standard(none)` is specified if no standardization is needed.

Examples

Without more options, `sqom` performs OM between each sequence and the most frequent sequence in the dataset, setting the indel costs to 1 and the substitution costs to 2. It standardizes the distances by dividing each distance by the length of the longest sequence in the dataset.

The results of the comparisons are written into the newly generated variable `_SQdist`, which can be used for further analysis, such as ordering a sequence index plot.

```
. sqom
Distance Variable saved as _SQdist
```

However, since many of the sequences are equidistant to the reference sequence, one may want to combine the variable with a second sorting variable. For figure 5 we have combined the distance with variables containing the length of each element (see section 3.3):

```
. egen length2 = sqlength(), element(2)
. egen length3 = sqlength(), element(3)
. egen length4 = sqlength(), element(4)
. egen length5 = sqlength(), element(5)
. egen plotorder = group(_SQdist length1 length2 length5 length3 length4)
. sqindexplot, order(plotorder)
```

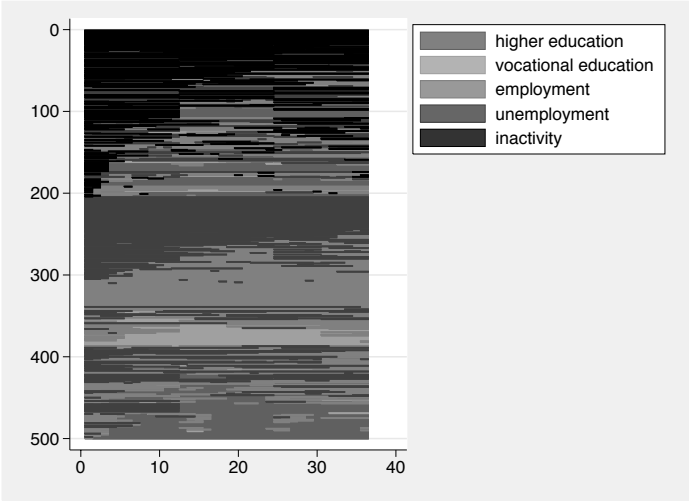

Figure 5: Sequence index plot with order from OM

In our next example, we use a full-substitution cost matrix, which was defined beforehand with standard matrix commands ([P] **matrix define**). Since the sequences in the data contain up to five elements, the subcost matrix needs to be a 5×5 symmetric matrix, where the first row/column refers to the element with the lowest category value. The name of the subcost matrix is inserted into the `subcost()` option. In addition to specifying a subcost matrix, we have also used the `refseq(15)` option, meaning that all sequences are compared against the sequence with `id==15` instead of the most frequent one. The results are written to the new variable, `om1`.

```
. matrix sub = 0,2,3,2,4 \
>              2,0,1,1,5 \
>              3,1,0,1,0 \
>              2,1,1,0,2 \
>              4,5,0,2,0
. sqom, subcost(sub) name(om1) refseq(15)
Distance Variable saved as om1
```

The running time of our second example increased because of the specification of the subcost matrix but can be reduced by applying the `k()` option. In our next example, we use almost the lowest possible value, `k(2)`, which does no harm in this case; i.e., the results are equal to the exact solution. However, there is no guarantee that this will always be the case.

```
. sqom, subcost(sub) name(om2) refseq(15) k(2)
Distance Variable saved as om2
. summarize om1 om2
```

Variable	Obs	Mean	Std. Dev.	Min	Max
om1	18000	.7437778	.5602692	0	1.972222
om2	18000	.7437778	.5602692	0	1.972222

Finally, we show an example with the `full` option, which requests that every possible comparison be calculated. With 500 observed sequences in the dataset, 124,750 distances need to be calculated. To reduce this enormous task, `sqom` performs the calculations only for the 347 *different* sequences, which still requires that 60,031 distances be calculated. Because computation time increases quadratically with the sequence length and the number of observations, you should expect long running times when specifying `full` with large datasets. For the following example, our computer (Pentium 4 with 2.66 GHz, Stata 9.2 for Linux) needed around 1 minute.

```
. sqom, k(2) full standard(longer)
Perform 60031 Comparisons with Needleman-Wunsch Algorithm
Distance matrix saved as SQdist
```

Naturally, the results of `sqom` with `full` cannot be stored as a variable in the existing dataset. It is therefore stored in the Stata matrix `SQdist`, which can be further processed by standard Stata commands and some specialized SQ-Ados (see section 6). Matrices are not stored with the dataset. The user-written command `mstore` by Michael Blasnik can be used for this task, however.

6 Grouping

On the basis of the distances calculated by OM, similar sequences might be grouped together. The step is straightforward if OM was performed on a reference sequence, implying that the generated variable represents the similarity of each sequence with the reference sequence. The variable can be grouped by applying standard Stata commands, such as `xtile` or `recode`, or by using `generate` with functions like `inrange()`, `inlist()`, `autocode()`, and `recode()`. It is even sensible to not group the similarity variable at all, as in figure 5.

However, when you are performing OM on a sequence-by-sequence basis (i.e., `sqom` with the `full` option), the grouping step is indispensable. Cluster analysis is the most common technique for this step. A variety of methods for cluster analysis on a dissimilarity matrix are available with the official Stata command `clustermat` (see [MV] **clustermat**). All these methods can be applied to the dissimilarity matrix saved by `sqom, full` as well.

There is one trap in applying the `clustermat` command to the dissimilarity matrix created by `sqom`, which stems from the fact that the sequence data and the dissimilarity matrix have different dimensions. Besides user-specified `if` or `in` qualifiers, the dimensions of the dissimilarity matrix produced by `sqom` depend on the sequence concentration and on the number of sequences with gaps. The dissimilarity matrix cannot be attached to the sequence data on a row-by-row basis, which also applies to the results from the cluster analysis of the dissimilarity matrix. The SQ-Ados therefore contain a command that helps to link the results of the user-specified `clustermat` command to the original sequence data. Its syntax is

`sqclusterdat` [`, return keep(`*varlist*`)`]

Without the `return` option, the command constructs a dataset, which is built from instructions left over by the last `sqom` command. The user may specify arbitrary `clustermat` commands, as well as applicable `cluster` postestimation commands in this dataset. After performing the cluster analysis `sqclusterdat, return` merges the cluster results with the original sequence data.[7]

We now provide an example of the entire procedure, performing two different cluster analyses and drawing a dendrogram for the cluster analysis by using Ward's linkage. Finally, the results of both cluster analyses are attached to the original sequence data.

```
. sqclusterdat
. clustermat wardslinkage SQdist, name(wards) add
. clustermat singlelinkage SQdist, name(single) add
. cluster tree wards, cutnumber(20)
. sqclusterdat, return
```

You must use the `clustermat` option `add` to allow `sqclusterdat, return` to merge the cluster results with the original sequence data. If you accidentally use `clustermat`'s `clear` option, `sqclusterdat` will revert to the original sequence data without merging the cluster results.

At the end of the process, the sequence data contain the variables produced by the cluster analysis. The variables suffixed with `_hgt` can be used in the same fashion as the distance variable produced by OM on a reference sequence. We use it to produce yet another version of the sequence index plot (figure 6).

```
. egen plotorder2 = group(single_hgt length1 length2 length5 length3 length4)
(648 missing values generated)
. sqindexplot, order(plotorder2)
```

7. A convenience command, `sqclustermat`, performs the three steps with one command. Cluster postestimation commands do not work in this case, however.

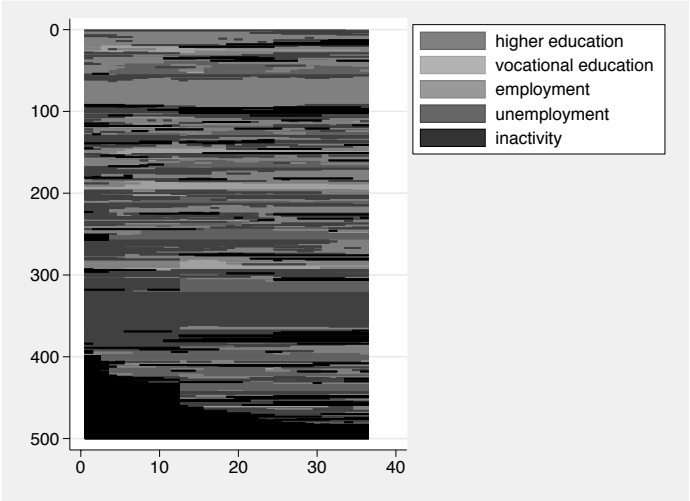

Figure 6: Sequence index plot with order from OM with option `full`

7 Appendix

7.1 The Needleman–Wunsch algorithm

Consider two vectors, R and C, that contain two different sequences of arbitrary length. Let m denote the length of R and n denote the length of C.

We start by constructing the $(m+1)\times(n+1)$ matrix L (the Levenshtein matrix) and initialize each cell with a zero. The cells of the first row and the first column of L are then filled with

$$\begin{aligned} L_{1,i} &= L_{1,i-1} + d; \quad i = 2,\ldots,m \\ L_{i,1} &= L_{j-1,1} + d; \quad j = 2,\ldots,n \end{aligned}$$

where d is the indel cost.

After initialization, the value of each cell $L_{i,j}$ ($i = 2,\ldots,m$ and $j = 2,\ldots,n$) is computed using the following recursive formula:

$$L_{i,j} = \min\left(L_{i-1,j-1} + s_{i,j}, L_{i-1,j} + d, L_{i,j-1} + d\right) \tag{1}$$

where $s_{i,j}$ is the substitution cost between the elements that are found in the sequences at the positions i and j, respectively. The unstandardized minimal distance between sequence R and C is in the cell $L_{m,n}$.

The algorithm was developed by Needleman and Wunsch (1970). It is an example of solving an optimization problem by dynamic programming and is guaranteed to find the minimal distance between two sequences.

7.2 Speed of sqom

The running-time complexity of the Needleman–Wunsch algorithm is $O\left(n^2\right)$, where n is the length of the sequences. To compute the distance between all N sequences, the algorithm has to be executed $N\times(N-1)/2$ times. Therefore, the running-time complexity of `sqom` grows quadratically with both the length of sequences and the number of sequences being compared. If a substitution cost matrix is defined, the next factor that influences the running time of `sqom` is the number of elements (p) that constitute the sequences. For every comparison of items in two sequences, the program has to search for the appropriate substitution cost in the substitution cost matrix, implying the worst case of linear complexity $O\left(p\right)$ for every single comparison of elements. Several precautions were taken to decrease the running time of `sqom`.

First, comparison to a reference sequence is the default.

Second, the Needleman–Wunsch algorithm is applied only to different sequences. This precaution implies a little overhead for data screening, which is worthwhile only if there is a reasonable concentration of sequences in a dataset.

Third, the `k()` option excludes some alignments from consideration. With `k()`, only that part of the Levenshtein matrix where

$$\left|\frac{i}{m+1}-\frac{j}{n+1}\right| \leq K\frac{1}{\sqrt{2}}\frac{\frac{m+1}{n+1}+\frac{n+1}{m+1}}{\sqrt{(m+1)^2+(n+1)^2}}$$

is calculated. If R and C have the same length, the program will explore the part of the Levenshtein matrix where the absolute difference between the horizontal and vertical index is less than or equal to K ($|i-j|\leq K$) (Kruskal and Sankoff 1983); i.e., the cells outside the middle region of the matrix are ignored. Practically, the `k()` option restricts the number of subsequent insertions/deletions that are allowed in the alignment of two sequences. The program will not find the minimum distance if the optimal alignment takes more than K subsequent insertions/deletions at some point. Thus using option `k()` implies no restriction if K is as large as n.

Finally, substitution costs are taken from the substitution cost matrix by using a so-called hash table, which is implemented as a vector of forward-linked lists. We solved a collision by chaining: all elements that hash to the same slot are inserted in an adequately linked list. Although searching for an element in the hash table can take as long as linear searching, under reasonable conditions the expected complexity is $O(1)$. Specifically, you should avoid the following features when constructing the set of your elements to assume constant complexity:

- The difference between two elements of sequences should not be multiples of 6,709, which is the principal number that we use for division when creating the hash function and is the length of the hash table.

- Avoid decimals in the set of sequences. The element is always rounded off by hashing. If a pair of decimal elements has the same integer part, the transformation is not injective and they will be inserted in the same cell of the hash table.

7.3 Limits

The memory complexity of the Needleman–Wunsch algorithm is $O\left(n^2\right)$. The algorithm is programmed with Mata, so the maximum sequence length is restricted to 2,147,483,647. Given that the human genome has around 3,000,000,000 base pairs, this limit imposes certain restrictions. There are, however, more severe restrictions imposed by Stata that apply to all SQ-Ados.

For the SQ-Ados, sequence data are expected to be in long format, which imposes no restrictions on sequence length. Much of the programming within the SQ-Ados is, however, done in wide format, so that the maximum sequence length is somewhat less than the number of variables allowed in the respective flavor of Stata (32,000 in Stata/SE and 2,047 in Intercooled Stata).

The command `sqom` with the `full` option stores its results by coercing a Mata matrix into a Stata matrix. The maximum dimension of that matrix is 11,000×11,000. Although one cannot manipulate this matrix with Stata's matrix commands, one can still perform a cluster analysis on the matrix regardless of the flavor of Stata and the setting of `matsize`.[8]

Given the limits and speed problems, the optimal matching as it is implemented in `sqom` seems capable of working with a moderate number of relatively short sequences. It has been tested using around 2,000 sequences with a maximum length of 100 positions.

8 Acknowledgments

Ben Goodrich made the text readable for native English speakers. Kenneth Higbee clarified what can (and cannot) be done in Stata with matrices created in Mata and placed back in Stata. We thank Richard Gates, William Gould, and Phil Schumm for providing help on Statalist. Ekaterina Selezneva pointed out bugs in previous versions of the program. Many thanks to all of them.

8. Also see http://www.stata.com/support/faqs/mata/matsize.html.

9 References

Abbott, A., and J. Forrest. 1986. Optimal matching method for historical sequences. *Journal of Interdisciplinary History* 16: 471–494.

Brüderl, J., and S. Scherer. 2004. Methoden zur Analyse von Sequenzdaten. *Kölner Zeitschrift für Soziologie und Sozialpsychologie* Sonderheft 44: Methoden der Sozialforschung: 330–347.

Brzinsky-Fay, C. 2006. Lost in transition: Labour market entry sequences of school leaves in Europe, Discussion Paper SP I 2006-111, Wissenschaftszentrum Berlin. Discussion Paper SP I 2006-111, Wissenschaftszentrum Berlin.
http://skylla.wz-berlin.de/pdf/2006/i06-111.pdf.

Cox, N. J. 2004. Stata tip 12: Tuning the plot region aspect ratio. *Stata Journal* 4: 357–358.

Diggle, P. J., K.-Y. Liang, and S. L. Zeger. 1994. *Analysis of Longitudinal Data*. Oxford: Oxford University Press.

Dijkstra, W., and T. Taris. 1995. Measuring the agreement between sequences. *Sociological Methods and Research* 24: 214–231.

Kogan, I. 2003. A study of employment careers of immigrants in Germany. Mannheimer Zentrum für Europäische Sozialforschung: Arbeitspapier Nr. 66.

Kohler, U. 2002. *Der demokratische Klassenkampf. Zum Zusammenhang von Sozialstruktur und Parteipräferenz*. Frankfurt a.M u. New York: Campus.

Kohler, U., and C. Brzinsky-Fay. 2005. Sequence index plots. *Stata Journal* 5: 601–602.

Kruskal, J. B., and D. Sankoff. 1983. An anthology of algorithms and concepts for sequence comparisons. In *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, ed. D. Sankoff and J. Kruskal, 265–310. Reading, MA: Addison–Wesley.

Levenshtein, V. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10: 707–710.

Macindoe, H., and A. Abbott. 2004. Sequence analysis and optimal matching techniques for social science data. In *Handbook of Data Analysis*, ed. M. Hardy and A. Bryman, 387–406. London: Sage.

Needleman, S., and C. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48: 443–453.

Rohwer, G., and U. Pötter. 2005. TDA's user manual.
http://www.stat.ruhr-uni-bochum.de/pub/tda/doc/tman63/tman-pdf.zip.

Sankoff, D., and J. B. Kruskal. 1983. *Binary Codes Capable of Correcting Deletions, Insertions, and Reversals*. Reading, MA: Addison–Wesley.

Scherer, S. 2001. Early career patterns: A comparison of Great Britain and West Germany. *European Sociological Review* 17: 119–144.

Stovel, K. 2001. Local sequential patterns: The structure of lynching in the deep south, 1882–1930. *Social Forces* 79: 843–880.

Waterman, M. 1995. *Introduction to Computational Biology*. London: Chapman & Hall.

About the authors

Christian Brzinsky-Fay is a political scientist at the Wissenschaftszentrum Berlin (Social Science Research Center). He is interested in labor market policy and education and is currently working on his dissertation with an application of sequence analysis for labor market data.

Ulrich Kohler is a sociologist at the Wissenschaftszentrum Berlin (Social Science Research Center) who has used Stata for several years. His research interests include social inequality and political sociology. With Frauke Kreuter, he is author of the textbook *Data Analysis Using Stata*.

Magdalena Luniak graduated in sociology from Warsaw University, where her focus was the application of mathematics in sociology. She now studies information technology at the Technical University of Berlin.