The following assumes that the reader is familiar with the broader outline of my proposed project as set out in my Oxford application-documents, specifically the main research proposal and/or either of the two UKRI funding application forms submitted alongside it.

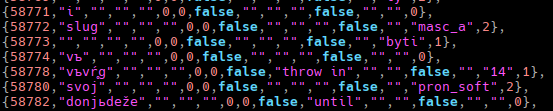
The stuff is largely unchanged since it was first done in 2021, when I was very new to programming, so as code it is pretty atrocious, but I've done a lot more foreign-language-focused programming since then so would easily be able to redo it more sensibly.

Why do I need an Autoreconstructor in the first place?

Since my goal is to enable quick, easy and comprehensive analysis of manuscript spellings, I need to produce LCS reconstructions of every form in the texts such that they will show up in searches for specific LCS phonemes or phoneme-class sequences (a crude search-mechanism for the already-reconstructed texts can still be found at [http://slavtexts.infinityfreeapp.com](http://slavtexts.infinityfreeapp.com/search.php)).

Doing this by hand isn’t really that much effort, but since the TOROT corpus contains high-quality lemmatisation and 10-place morphology-tagging, it’s more fun to take the list of TOROT lemmas, reconstruct for each the invariant part of the stem[s], label each one for inflectional-category, then just stick the correct inflections onto the end of the correct stems according to what the morphology tag tells us.

To hold this and a lot of other possibly relevant information (loanword-origin and source-language, whether something is a blatant non-integrated post-LCS loan, whether a differently-formed doublet of the stem occurs which TOROT has no separate lemmatisation for, etc.), I have a sort of ‘master-spreadsheet’ (<https://docs.google.com/spreadsheets/d/1qw75jk_oLbjohmdh_qlC6B1yb9xRhwqY>) which started life just as a list of all the TOROT Church-Slavonic lemmas. To fully reconstruct the Marianus and (TOROT’s part of) Euchologium Sinaiticum, I just filtered the lemmas to only display those which occur in Mar. and Euch., then manually reconstructed the stems and added the various pieces of auxilliary information. The necessary parts of the spreadsheet then get converted into a C++ std::set data-structure of what I’ve called Lemma structs, and it’s from this std::set that the program retrieves the correct Lemma struct and inspects its stem-form and inflection-class fields to start producing an LCS form.[[1]](#footnote-0)

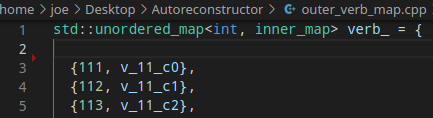


If you look at the third row you can see that the stem of byti is blank and its inflection-class is ‘byti’, because it essentially has its own whole paradigm. Indeed, the number of separate inflection-classes is pretty obscene for something which claims to be “automatic”: I counted 47 verb-classes and 40 nominal-classes (and I haven’t even added \*dǫti yet because it doesn’t occur in Mar. or Euch.), but there are actually more class-names than inflection-tables because, for example, all o- and a-stem nouns and all hard-stemmed adjectives get passed to the same list of 63 (3 numbers \* 7 cases \* 3 genders) endings, and similarly I reuse the pronoun \*jь’s table for the long-adjectives and participles.

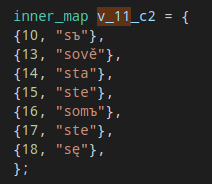
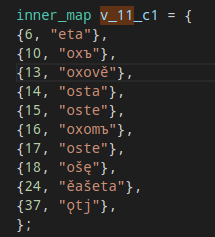
The way I read the morphology-tag is very boring and not really relevant, but if you want to see it you can just look at the massive switch-statement at the top of this file <https://github.com/12401453/Autoreconstructor/blob/main/changeLemma_field.h>. From a numerified-version of the morphology-tag I compute a number that corresponds to the row of the inflection-table that the given form should select its endings from, e.g. for a verb it will be between 1 and 44, while for a nominal-class it’d be between 1 and 63 (though gender-specific nominal classes would only use a third of the 1-63 range, e.g. the table for the feminine R-stem nouns only has entries from 22-42). Which inflection-table to look into is determined by the inflection-class label which is contained in the massive std::set of Lemma structs as described above.

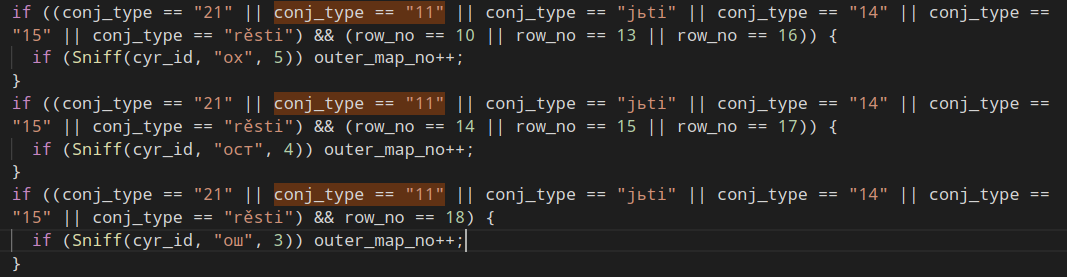
What about morphological innovations? Case study: class 1 verbs

My inflection-classes 11, 14 and 15 correspond to subsets of Diels (1963:244-251)’s class 1.1 verbs, i.e. these are class I verbs which add their endings directly to a consonantal-stem with no intervening jot. They differ from my classes 12 and 13, which are respectively nasal and liquid-stemmed, in that their 2nd and 3rd sg. bare aorists add \*-e rather than nothing: \*nes**e** vs \*prijьn > \*priję > (in OCS also with a -tъ 3rd sg. marker added) приѩ[тъ], \*umer > оумрѣ[тъ]).

I store the inflection-tables in a C++ std::unordered\_map container, which is basically a list of key-value pairs[[2]](#footnote-1), and for each inflection-class’s key I use integers that end in the number ‘1’ and are at least 10 away from the next class’s key. The inflection-table pointed to by the key ending in ‘1’ stores the ‘correct’ LCS endings, and then deviant endings are stored in supplementary tables pointed to by keys which are subsequent integers.

For example, above we can see the tables for the class 1 verb endings (classes 11, 14 and 15 only): the table indexed by the key 111 contains the ‘bare’ aorists such as those found in Marianus, like 3rd pl. падѫ, 3rd sg. несе, 1sg. придъ, while the second table (key 111) contains the secondary S-aorists in \*-ox- (Zogr. падошѧ etc.), then the third table (key 113) contains primary S-aorist endings, to produce things like Mar. 1sg. рѣхъ, 3sg. възнѣсѧ:

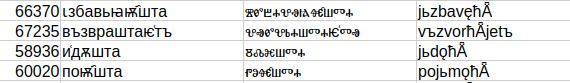
  
The real dirt and grime of the Autoreconstructor’s current extremely amateurish implementation is in how these different morphological alternatives are detected: I essentially just have banks and banks of if-statements that inspect the last few letters of a cleaned-up version of the TOROT form, if the inflection-class and inflection-table row-number (as computed from the morphology-tag) correspond to forms which often get replaced in my target texts (hence currently it only tries to detect OCS-specific deviances).



Here you can see the detection of (among others) these innovated class 1 aorist forms: all the Sniff() function does is chop off the specified number of letters from the end of the Cyrillic form and check to see if it contains the letters passed to it as its second argument[[3]](#footnote-2). If it does, then Sniff() returns ‘true’ and the integer-key of the current inflection-table is incremented by one (111 + 1 = 112), causing it to instead select the deviant endings stored in the supplementary table.

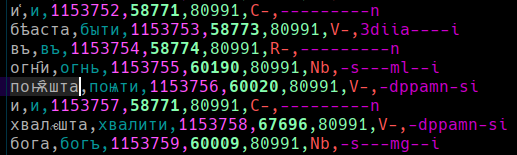
**Annotation-mistake detection**

Here we can see a random example of two bad autoreconstructions caused by incorrect TOROT lemmatisations in the Suprasliensis (ignore the Glagolitic):



In the first instance TOROT must simply have \*jьzbaviti rather than \*jьzbavĺǞti, which is understandable given that the imperfect tense of these class 4 verbs is indistinguishable from their secondary-imperfective class 5.4 counterparts (indeed that’s likely where this type of secondary-imperfective came from), but this is a present active participle so the lemmatisation should be that of the secondary-imperfective class 5.4 \*jьzbavĺǞti.

The second instance is a more extreme mistake, since we have ‘sing’ lemmatised as ‘grab’, as is clear from the context[[4]](#footnote-3) :



If we converted these erroneous reconstructions to the regular but unattested ‘normalised’ OCS system which is sometimes employed, i.e. избавѧща, поимѫща (a trivial task), then the “edit-distances” between the normalised-OCS forms and a cleaned-up version of the Suprasliensis forms should easily be large enough to pass whatever threshold we might set for alerting us of possible TOROT mistakes.

A potentially cool side effect of having such an Autoreconstructor is it could be combined with automatic lemmatisation and morphological-tagging (which Hanne can already do surprisingly effectively and which I think could possibly be improved further by more careful and targeted data-cleaning and potentially also neural-network approaches like the one described here [Morpheus reference]), to produce an LCS reconstruction straight from a raw text, potentially even a raw manuscript if Transkribus can be trained well enough (though that probably is pushing our luck).

A student should be able to point their phone at a past exam-paper or homework commentary-passage and get given back a bullet-point list of all the relevant phonological features in the text such that they can completely cheat; if I am successful in producing high-quality comprehensive phonological-annotation then all that would be necessary to achieve this is OCR good enough to locate the printed passage in my database, but it would be interesting to see if such cheating could also be enabled using nothing but the raw text.

The size of the corpus of relevant texts is small enough, though, that automated techniques are only ever envisaged as a starting point for manual correction: I don’t even trust Jagić’s editions and fully intend to read every single text I study (with the possible exception of Psal. Sin. which is a nightmare) cover-to-cover from manuscript-photographs before I sign anything off.

As a more computationally-competent but less linguistically-rigorous example of what is possible with databases and browser-technologies you can look at <http://57.128.170.157:5000/text_viewer>, which is a VPS-instance of an application to help serious language-learners learn vocab by reading and annotating texts. Many of my texts there are Danish, which as you’ll know is full of Germanic separable-verbs and multiword-idioms and identical forms that belong to different lemmas and multiple homomorphic lemmas, and it can deal with all of it in an efficient and scalable way, so problems such as how to display / encode phonological phenomena which span multiple lexical words (изд рѫкъі, Psal. съмѩ сѩ <\*sъmęsъ\_\_sę etc., clitics basically) will not be that hard to get around.

because those problems often point to significant events in the history of the language that you would never even think about

1. Compiling all this data directly into the C++ program (or using C++ at all for this task) rather than reading it off of disk at runtime is an extremely stupid way of doing things and can cause problems with compilation. [↑](#footnote-ref-0)
2. Look-ups in a map are what they call constant time-complexity (O(1)), meaning it’s just as fast no matter how many key-value pairs the map contains, because rather than test each key for equality with the looked-for key until you find it, they use a hashing-function which converts the key to a pointer directly to that key’s value-data, meaning the lookup-time is always just equal to the time taken to compute the hash. The same thing happens when you have an Index on a database-table column, which is why looking things up by a table’s PRIMARY KEY column is fast even with millions of rows in your table, whereas lookups by non-indexed rows get progressively slower as the table grows. [↑](#footnote-ref-1)
3. It actually just chops off the specified number of bytes (with each Cyrillic character (usually) being 2-bytes long in UTF8), because I chose in my infinite wisdom to use a language with no inbuilt Unicode support to make a program that does nothing but non-ASCII string-manipulation. [↑](#footnote-ref-2)
4. Note also TOROT’s baffling sporadic replacement of Supr. ꙙ by the hooked Glagol. nasal ⱕ, which actually only occurs in Zogr. and Mar. (ⰳⱃⱔⰴⱕⰻ, ⱀⰵⱄⱕ etc.) and which Kortlandt (1979?) says is a remnant of \*y̨ corresponding to the NSl. \*-a endings on masc. Nsg. hard-stemmed class I PRAPs. Olander 2015:89-90 has more discussion, though beware his oversimplifying characterisation of it as “*ę* with a special sign” (Велчева and Trubetzkoy are crying out in pain). [↑](#footnote-ref-3)