

# Automatic Back-transliteration of Romanized Bulgarian

---

*Aleksandar Savkov*

*International Studies in Computational Linguistics*

*aleksandar@savkov.eu*

## 1 Introduction

Automatic transliteration (AT) is one of the unpopular, but rather important fields in computational linguistics (CL). It has a very diverse and language-specific set of problems, which renders its generative methods more often ad hoc than general. In this paper we present a small project on back-transliteration of Romanized Bulgarian using hidden Markov models (HMM).

The specific case that we will discuss is a phenomenon that appeared in the Internet era. A great deal of text resources in Bulgarian is available on the web, but unfortunately written with Roman characters. Back-transliterating such texts could be beneficial for both cross-language information retrieval and also for everyday users who cannot type in Cyrillic (which led to the creation of these huge text resources in the first place). We also present all official systems used for transliteration as well as the unofficial one used by most people on the Internet. The two main problems that we need to resolve there are the character compound recognition and ambiguous character transliteration.

Hidden Markov models have been used in transliteration before, for example (Kashani, 2007), who use the Viterbi algorithm in two runs to transliterate English named entities (NE) from Arabic back to English. In this paper we show how HMM-driven automatic transliteration methods improve the results of lexically-driven ones, when implemented together.

## 2 Goals

After the great success of statistical search engines on the web, we may consider cross-language information retrieval to be the next frontier. The role of automatic transliteration in the field of information retrieval (IR) is commonly related to the role of named entity recognition (NER), which has been commonly implemented through equivalence classes. For example, {*Opel, Опел*} is an equivalence class that will unify the results for queries about the automobile company written with both Roman and Cyrillic characters. However, this is an imperfect solution requiring constant updates and corrections. It performs even poorer when applied to words or phrases without a counterpart in the non-query language, e.g. matching *garbage collector* to *зарбидж колектор*. The increasing demand for better quality and wider coverage of IR services impose the need for a better and automated way of transliteration.

And while in the example we just gave there is an alternative method for achieving some accurate results, the case we present in sections 3.1 and 3.2 strictly requires automatic transliteration. Back-transliterating from Romanized Bulgarian is a task that easily allows results with accuracy over 50% or 60%, but the real challenge is to achieve perfect or nearly perfect results. The nature of the applications of such an implementation as ours also demands higher speed. This being said, we want

to be able to provide a reliable and fast transliteration usable in two directions: transliteration used in automated analysis of queries and simple transliteration of short texts that could be helpful to web users who are not able to type in Cyrillic or merely want to transliterate a Bulgarian text written with Roman characters. These applications need considerably fast software where a plain approach checking all possibilities in a dictionary is not enough. To make this possible we suggest using HMM methods for computing the most likely transliterations and then confirming their existence in a lexical resource. We also suggest using the HMM based results to increase the transliteration capabilities beyond the threshold set by the available lexical resources.

### 3 Transliteration issues in Bulgarian

Transliteration in general is the sum of all specific problems of the various language-alphabet pairs. This means that one does not have to and practically cannot address all possible issues in this area when dealing with one language pair. However, the most prominent problems that any automatic transliteration should tackle are directly inferred from the two ways of transliterating: using pronunciation as a bridge or using direct character matches between the alphabets. Using the additional state of sound representation is a more sophisticated way that also requires much more human knowledge. This process is no exact science and often has different results when performed by different human transliterators. On one hand it has the appeal of translating words and producing accurate loan word transliterations that could be easily recognized by a human reader in other languages (because of the very similar pronunciation); on the other hand, it requires additional linguistic resources for its automation – a pronunciation model or a pronunciation dictionary. Transliteration processes, that use character matching, depend highly on the number of loose (one character matching more than one character in the other alphabet) matches between the character sets and thus the complexity of such tasks varies to both extremes. The case that we are presenting in this paper makes use of the latter approach and has a rather high complexity, the reasons for which we explain in the following section.

#### 3.1 Language and alphabet setting

The many one to one matches between Roman and Cyrillic characters suggest that the complexity of the task should not be too high. Dealing with one language in two alphabets seems simpler than dealing with two languages and two alphabets. Nevertheless, there are circumstances in this case that increase the complexity significantly.

We use transliteration by character matching as our starting point in this setting. The challenges with this method come on two levels. On the first level are the most obvious differences in character count and character correspondence. They draw up the two base issues of Cyrillic to Roman<sup>1</sup> transliteration: some characters do not have correspondent characters (e.g. ж, џ, њ, љ, ѝ) and some characters have more than one (e.g. в – v, w; љ – j, y; б – j, y; њ – a, u). Furthermore, some characters are transliterated with a combination of characters (e.g. ж, џ). These problems are refined on the second level where they form four types of characters: one character matching many single ones (e.g. в – v, w), one character matching one character combination (e.g. ж - zh), one character matching many character combinations (e.g. џ – sht, scht), one matching at least one character and

---

<sup>1</sup> The Roman set of characters used in ASCII

at least one character combination (e.g. *u – ts, c*). All these problems could be easily avoided when transliterating to another alphabet, because a transliterator can choose one scheme and be consistent, but the solution to them greatly affects back-transliteration, which is the focus of our work in this paper.

### **3.2 Back-transliteration**

In the previous section we showed the four types of characters based on the issues they present with regard to transliteration. To deal with them, however, we need to solve two main problems: we need to recognize character compounds (e.g. *sh* resulting into *w* vs. *cx*) and we need to find the most appropriate back-matching character pairs (e.g. *a* resulting into *a* vs. *ѡ*). These tasks are not trivial and one may need to consider trailing as well as preceding character context and sometimes even word context. If the transliteration system used for Romanization is known before hand, back-transliteration is much simpler: it comes down to character compound recognition. However, we are aiming not only at transliterating without any prior knowledge of the system used, but we also want to be prepared for mixed or improvised systems.

#### *Transliteration systems*

Numerous transliteration systems have been used through the years (see first eight systems in Table 1) – some based on French, some based on English and some based on other Slavic transliteration systems – but we are not interested in them separately. We are interested in the unofficial mix of all of them called *Metodievitsa*<sup>2</sup>, which is the last system listed in Table 1. It is a bit of an overstatement to call it a system, because it really is a collection of all transliteration possibilities that have roamed the Internet in the last ten or more years. The lack of standards and guidance in the early days of computers and Internet in Bulgaria has created this system of very loose rules that everyone uses in their own personal way and with almost no consistency. It also allows people to play with some letter correspondences and use not only sound similarities, but also orthographical similarities (e.g. Roman *y* is used for Cyrillic *y [u]* because they have the same orthographical symbol) or simply the same position on a keyboard layout (e.g. *y* is used for *ѡ* because they respond to the same key on QWERTY and Bulgarian phonetic layout). It is evident from Table 1 that this system makes things almost too complicated to solve and yet there are ways to do it.

#### *Foreign words and names*

Before moving on to our solutions we ought to make notice of the issues of loan words and foreign names. They have been a main focus of scientific work in the area of transliteration (e.g. (Kang & Choi, 2006) deal with back-transliterating English words from Korean, (Stalls & Knight, 1998) address back-transliteration of English names and technical terms from Arabic texts); however we will not put so much stress on foreign words at this stage of the development of our transliterator, because they are a small part of our target lexis. We want to postpone dealing completely with this issue, because in our alphabet setting such words have to be transliterated using rules and models for inter-language transliteration. For example, we can take the English word *server*, which is a loan word and we can take a look at its Cyrillic spelling *сѣрѡѡѣр*. It is based on its English pronunciation and thus back-transliterating with our set of rules does not work - *\*сѣрѡѡѣр* is incorrect. When Romanizing Bulgarian it is very common that such words are written in their original spelling, which is a problem for a transliteration program, because they need to be recognized and transliterated accordingly. The

---

<sup>2</sup> A good description of the system and its problems is given in (Savkov, 2009)

important goals of our project are pointed in another direction, so recognizing foreign words and then keeping them as they are will suffice at this point. Another fact that makes things a bit simpler when dealing with loan words is that most of the loan words that are commonly recognized as loan words and thus written in their original spelling are English, which are the newest ones in the Bulgarian lexicon, mostly acquired through the Internet.

## 4 Implementation

Our goal in our implementation is to introduce HMM methods into the setting of back-transliterating Romanized Bulgarian and so to increase the capabilities of only lexically-based transliterators; in addition we also tried to use adequate lexical resources to bring enough reliability. Those two independent methods were combined in order to achieve even better results in the end. Here we first explain how each of them works and then their combination in one algorithm.

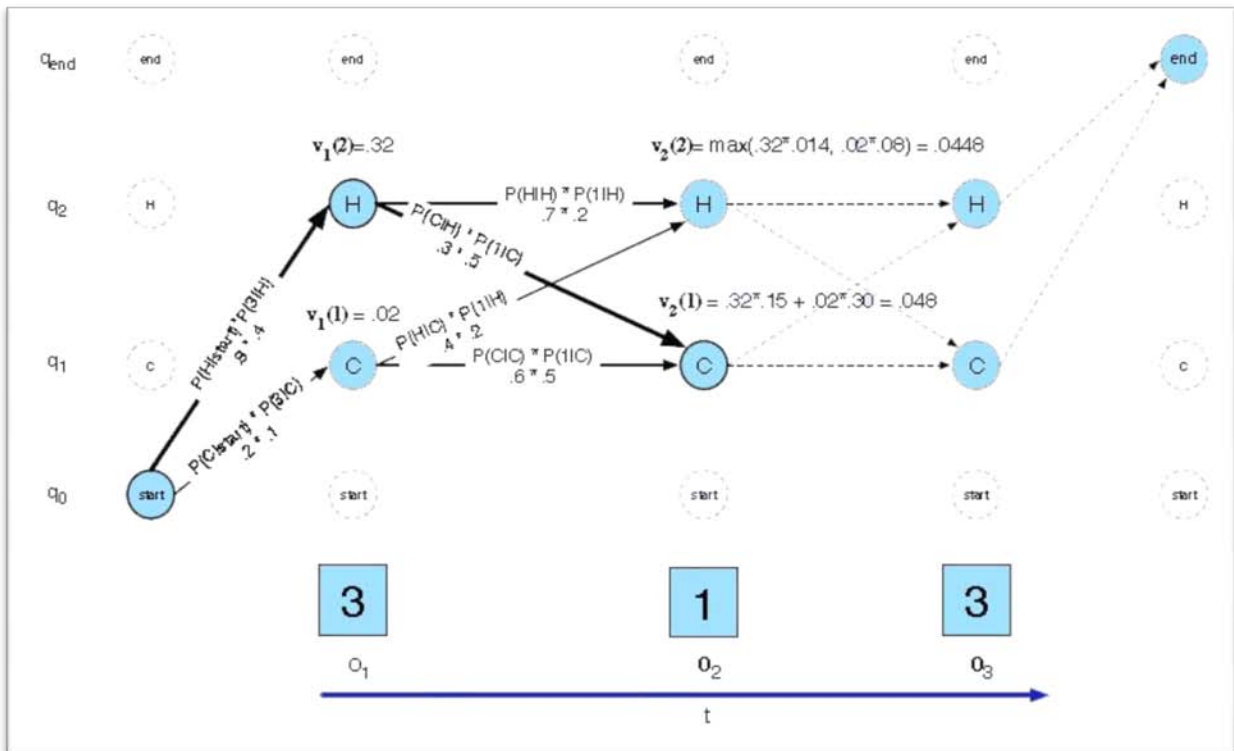
### 4.1 Hidden Markov Model

The most efforts in this study were put into optimizing lexical queries' performance with statistical methods. We generate the most probable transliterations and look them up in the dictionary instead of trying all possibilities. An advantage of this improvement of the lexical method is the fact that its suggested output is still the statistically most probable transliteration of the word. This expands the capabilities of our implementation beyond the reach of our lexical resources, providing a relatively good transliteration guess for any given word (even the ones not in the dictionary).

Before elaborating on the issues we faced while implementing an HMM solution to the transliteration problem, we want to recall section 3.2 where we pointed out the two fundamental problems of back-transliterating Romanized Bulgarian: recognizing character compounds and resolving character ambiguity. To explain better how these problems can be solved with HMM methods we will briefly recap an example from (Jurafsky & Martin, 2008) about Jason Eisner's diary and how it was used for weather prediction. The simple Jason example consists of observations being the count of ice cream balls that Jason ate and noted in his diary on a given day, the hidden states are the weather states: hot and cold. Presumably, Jason ate more ice cream on a hot day than he did on a cold one – this is the emission probability extraction. The other aspect, the aspect of transition probability, calculates the chances of a cold or hot day following a given (hot or cold) day. In Figure 1 is illustrated how the Viterbi algorithm finds the most probable path in a simple example case. However, that does not work out of the box for the problems of back-transliteration like it would have if there were no character compounds. In the hypothetical case of no character compounds, the observations are what the program gets as an input – Roman characters – and the hidden states are the Cyrillic characters. However, character compounds, even unambiguous ones, are unacceptable for the HMM trellis in which observations and hidden states correspond one to one. A very important point in our implementation is exactly the way we go around this obstacle and simulate character compound recognition. We will change some of the assumptions in Jason's example in order to relate to character compounds. Normally the assumption is that Jason made entries about ice cream once a day, but in we will suppose that he did not mark dates and distinction is made only between separate entries. This means that he may write about eating ice cream twice on the same day. Further, we add the extreme hot day hidden state on which Jason supposedly wrote about eating much more ice cream than usual or went twice to the ice cream stall and had the normal hot day

amount. So if Jason had written that he ate 3 ice creams we will know that it was either a hot day or an extremely hot morning. And if in the next entry he had written that he ate three more, there may have been either two hot days in a row or one extremely hot day – which one is it depends, of course, on the statistical model, but the important thing here is the possibility of recognizing such a sequence. The extreme hot days in the alphabet setting need to be simulated by meta-characters that will represent character compound parts. They are treated as hidden states and thus it is possible to calculate routes through the trellis involving two or more compound character parts, making recognizing the whole compounds possible.

Figure 1 Viterbi algorithm



An important dilemma that will be discussed a little later is the principal idea of using the same meta-character for all compound parts or different ones for each position in the compound. We decided in favor of using one character, because increasing the amount of meta-characters also increases the amount of calculations significantly. We may add to it that emission probabilities would be even harder to gain (see below more about how this is done).

Apart from developing the theory about using HMMs in our transliteration setting, we had to go through three levels of implementation to achieve results based on statistical calculations – trellis implementation, hand-crafting a statistical model and optimizing the HMM algorithms for our needs.

**Trellis implementation**

For the first stage of our implementation – the trellis - we used a Java implementation that was created for training transliteration models on a pet corpus (see more about the model below) using the forward-backward algorithm. It was originally designed to construct a trellis out of *State* and *TimeStep* objects and then calculate forward and backward probabilities as well as additional mid-calculation probabilities needed for the forward-backward algorithm. The trellis is meant to hold

structured information about each state in an accessible way. We implemented two things in order to be able to generate the best path using the trellis. First we provided the states with a way to hold information about Viterbi probabilities and second we implemented the Viterbi algorithm in such a way that it makes use of the trellis program as a data structure and operates with it.

### *Crafting a model*

When put in simple words, an HMM-based transliteration is as good as its statistical model. Its success depends on the accuracy and balance of its emission and transition probabilities. Acquiring both simultaneously, however, is not an easy task. Unfortunately we cannot use the forward-backward algorithm implementation on a full-scale training corpus, because while it is possible in theory, it is not manageable for our hardware at the moment. The best we could do was to create the model on our own.

Reliable transition probabilities are simple enough to be extracted from any big enough corpus. We have a newswire corpus of size 53MB that fitted our needs perfectly. Using it we calculated transition probabilities based on the relation of the count of a particular bigram towards the count of all bigrams. Each of these probabilities were, in fact, the probabilities of character A and character B occurring in this order, which in practice is the chance of a transition taking place from character A to character B. A small trick we had to use when doing this was first converting the whole corpus to our meta-character alphabet leaving out the original characters (e.g. *w* would be replaced by *\_w\_w*). Then we duplicate the results for all pairs of meta- and normal characters because otherwise the original characters are left as not occurring bigrams. We can do this, because in fact transition probabilities should not make a difference between meta- and original characters. To keep the ratios true we compensated the overall bigram count with the sum of all duplicated counts.

Gathering enough data for a reliable corpus-based training of the emission probability part of the model is impossible for such a small-scale project. For the training one needs a character-aligned bi-alphabetical corpus, which can be produced only by colossal manual labor. We have created a small pet corpus to use for testing purposes, but its 2000 words are hardly enough for reliable training, thus an emission probability model was hand-crafted. It was not a single action but rather a process that was still developing as we tested our program and should also be further improved.

### *Optimization*

The HMM algorithm was optimized in one important way. We introduced a validation mechanism for the cases of meta-characters. It is a solution to a problem and an optimization at the same time. The way we go around character compounds allows their recognition, but unfortunately malfunctions too often. We relied too much on the transition probabilities to control the appearance of meta-characters and also to restrict them to appear in all character compound candidate time steps or none at all. Their influence was overestimated, which called for an on-the-fly validation of possible transliterations. On the upside the validation saves some time for the calculations, because it sets some of the paths directly to zero probability. On the down side, it makes a considerable amount of checks for each compound capable character.

## **4.2 Lexical resources**

The only way to absolutely confirm a transliteration remains looking it up in a dictionary. Thus in this section we will briefly present our lexical resources. We gathered 265 911 different Bulgarian word forms for our dictionary, which was based on a small Bulgarian dictionary. We expanded it by

scraping all different forms from twenty Bulgarian novels and the newswire corpus that we also used for calculating transition probabilities. In section 3.2 we mentioned foreign words with regard to our task and to what extent they are handled by our implementation. To recognize them we use a small dictionary (505 words) of English technical terms and words used in the IT sphere.

In addition we tried to amplify the dictionary's recognition capabilities by using an open source Bulgarian stemmer<sup>3</sup> called BulStem. We supposed that by stemming unknown word forms we could recognize them and compensate for not having all possible inflectional forms of all the words in our dictionary. The stemmer was not included in the final version of the implementation, because we established that in the setting of our task it malfunctioned very often. And by malfunctioning we mean that it produced non-existent stems.

### **4.3 Transliteration algorithm**

Having good methods for transliteration is only half the job done. How these methods are applied, in what order and in what cases, are questions that hold the key to the other half of the successful transliterator that we are trying to produce.

The first step in our transliteration algorithm is generating the Viterbi-based transliteration and looking it up in the dictionary. We decided that this should be the first step, because it theoretically provides the most probable transliteration and if it is a valid one the cost of the whole process will be the Viterbi path generation and one dictionary look-up, which is fairly low.

If the dictionary look-up fails, however, the next step is to use the trellis to generate all possible transliterations and their probabilities, sort them and check if one of the ten best is in the dictionary. We do not have to elaborate on the technical details of this operation, because it is trivial in its essence. This step in practice enhances the dictionary look-up of all possible transliterations and limits its cost to ten word look-ups. Certainly, there are more efficient ways to generate all transliterations, but this method pays off when transliterating average-length and longer words, where the latter are, in fact, the ones usually not in the dictionary.

Finally, if those methods cannot find a transliteration in the dictionary, we use the failsafe step – the best guess of the Viterbi path transliteration.

## **5 Results and application**

Accurately testing the performance of a transliterator is a very difficult task for the same reasons as obtaining emission probabilities from a corpus. We already mentioned the small character-aligned corpus that was manually created. It has 1085 different word forms taken from three Wikipedia articles, slices of which were transliterated by different people. The testing results showed an accuracy of over 99% on character transliteration level (99.5% on the word level), but we are skeptical of this success. The Wikipedia articles contain mainly popular words required by the strict style of scientific writing. We expect lower accuracy when transliterating corpora gathered from forum posts or novels, for example.

---

<sup>3</sup> BulStem – developed by Preslav Nakov - <http://people.ischool.berkeley.edu/~nakov/bulstem/index.html>

From the mistakes that our transliterator makes, we can bring out three issues. Sometimes the model is not good enough and ranks the right transliteration below the first ten. We expect such problems to occur less with the graduate improvement of the model. Then there are rare compound words that are almost never listed in the dictionary. They are a natural weak point and also one of the target zones of the HMM methods – thanks to the statistical analysis these words are not transliterated in an inadequate way, but merely have one or two wrong characters. The third kind of errors, though, we cannot help. They are in Roman sequences that have two possible valid transliterations and in those cases there is no way of disambiguation without involving syntactical analysis (e.g. *koi* may be transliterated to both *κοῦ* /who-male-sg/ and *κου* /who-pl/).

We have also created a web interface<sup>4</sup> making use of our transliterator as a back end and offering the users the opportunity to transliterate small texts up to 2800 characters. Furthermore we are working on a small web plug-in that will allow the integration of the transliterator into any web page giving it the possibility to transliterate the text in any of its text areas by communicating with the Java servlet on our server.

## 6 Discussion

The software project presented in this paper was very successful within the small testing possibilities, but we do not consider its testing a closed chapter. The objective that we put most stress on in this project was achieving results from a functional integration of an HMM into the transliteration process. In addition to the results given in the last section, we present the distribution between Viterbi check-up and dictionary check-up in erroneously transliterated words (not characters): 62.5% dictionary-made errors (1.5% of all dictionary transliterated words) and 37.5% Viterbi-made errors (26.7% of all Viterbi transliterated words). These results show that the Viterbi algorithm actually improves the overall results, because the Viterbi algorithm transliterated correctly 73.3% of the words that were not found in the dictionary. This comparison, however, cannot induce that using HMM is the better method, because both methods were used for only some of the words of the testing corpus. The real accuracy of the Viterbi algorithm method is 82.2% when performing alone without dictionary, whereas the dictionary alone is 96.3% (both on word level). Considering these results, we can say that the statistical methods we integrated into our transliterator have gained us more than 3% of transliteration accuracy, which is around what we were hoping for. On the other hand we have to say that in numerous cases the statistical model had significant flaws that could definitely be improved in the future.

We are content with the achievements of our implementation, but there are still some serious improvements that need to be made before an application using our methods and resources could be called helpful in a real world IR project. The current speed of transliteration is about half a second per word. This is rather slow and it will increase as more lexical data is added to the dictionary. To deal with this we set developing a finite state acceptor in the way of (Knight & Graehl, 1997) as one of our major goals related to this project in the future. This will speed up the queries to the dictionary, which subsequently may change the order of transliteration methods and make trellis construction and calculations not always necessary as it is now, which in turn means possibly even more speed.

---

<sup>4</sup> Available at <http://aleksandar.savkov.eu/projects/bat/index.html>

We also want to make a remark with regard to the English words that we try to handle in the simplest way possible (by just excluding them from the transliteration process) in this project. In fact, these words lead to a completely different issue – back-transliterating foreign words. We will not go into detail of how this could be done, but we point out that such an undertaking could be built parallel to this project and also used by it.

Finally we conclude that based on the results we received from our transliterator we consider using both lexical and statistical methods more efficient than using either of them alone.

## **7 Bibliography**

Jurafsky, D., & Martin, J. H. (2008). *Speech and Language Processing*. Prentice Hall.

Kang, B.-J., & Choi, K.-S. (2006). Two Approaches for The Resolution of Word Mismatch Problem Caused by English Words and Foreign Words in Korean Information Retrieval. *Computer Processing of Oriental Languages* (pp. 85-96). Singapore: Springer.

Kashani, M. M. (2007). *Automatic Transliteration from Arabic to English and its impact on Machine Translation*. Simon Fraser University.

Knight, K., & Graehl, J. (1997). Machine Transliteration. *Proceedings of the Thirty-Fifth Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics* (pp. 128-135). Summerset, New Jersey: Association for Computational Linguistics.

Savkov, A. (2009). *Automatic Transliteration Focus on Bulgarian*. University of Tübingen, Department of Linguistics.

Stalls, B. G., & Knight, K. (1998). Transliterating Names and Technical Terms in Arabic Text. *Proceedings of the COLING/ACL Workshop on Computational Approaches to Semitic Languages*.

Wikipedia.org. (n.d.). *Транслитерация на българските букви с латински*. Retrieved May 12, 2009, from Wikipedia:

[http://bg.wikipedia.org/wiki/Транслитерация\\_на\\_българските\\_букви\\_с\\_латински](http://bg.wikipedia.org/wiki/Транслитерация_на_българските_букви_с_латински)

## 8 Appendix

Table 1 Romanization systems

Cyrillic	BGN/PCGN	ALA-LC	<u>Standard 1956</u>	BDS 1596:1973	UN 1977 (Andreychin)	BDS ISO 9:2001	Danchev System	<u>Streamline System</u> <sup>5</sup>	Metodievitsa
а	a	a	a	a	a	a	a	a	Aa
б	b	b	b	b	b	b	b	b	Bb
в	v	v	v	v	v	v	v	v	Vv, Ww
г	g	g	g	g	g	g	g	g	Gg, r
д	d	d	d	d	d	d	d	d	Dd, g
е	e	e	e	e	e	e	e	e	Ee
ж	zh	zh	zh (ž)	zh (ž)	ž	zh (ž)	zh	zh	Zh zh, Zz, Jj, Gg
з	z	z	z	z	z	z	z	z	Zz, 3
и	i	i	i	i	i	i	i	i	Ii, Uu
й	y	ï	j	j	j	j	y	y	Jj, Ii, Yy
к	k	k	k	k	k	k	k	k	Kk
л	l	l	l	l	l	l	l	l	Ll
м	m	m	m	m	m	m	m	m	Mm
н	n	n	n	n	n	n	n	n	Nn, H
о	o	o	o	o	o	o	o	o	Oo, 0
п	p	p	p	p	p	p	p	p	Pp, n
р	r	r	r	r	r	r	r	r	Rr, Pp
с	s	s	s	s	s	s	s	s	Ss, Cc
т	t	t	t	t	t	t	t	t	Tt
у	u	u	u	u	u	u	ou	u	Uu, Yy
ф	f	f	f	f	f	f	f	f	Ff
х	kh	kh	h	h	h	x (h)	h	h	Hh, Xx
ц	ts, t*s	t̄s	c	c	c	c	ts	ts	Cc, Ts
ч	ch	ch	ch (č)	ch (č)	č	ch (č)	ch	ch	Tsh tsh, Ch ch, Cc, `
ш	sh	sh	sh (š)	sh (š)	š	sh (š)	sh	sh	Sch sch, Sh sh, Ss, 6, [
щ	sht	sht	sht (š)	št	št	sth (š)	sht	sht	Scht scht, Sht sht, St st, 6t, ]
ъ	ŭ	ŭ	â	a (â)	ă	a` (")	u	a	Aa, Uu, Yy, 1
ь	'	j	j	` (')	j	y	y	y	Ii, Jj, Xx, b
ю	yu	iū	ju	ju	ju	yu (û)	yu	yu	Yu yu, Ju ju, Iu iu, Uu
я	ya	iā	ja	ja	ja	ya (â)	ya	ya	Qq, Ya ya, Ja ja, Ia ia
ѣ	e, ya	iē				ě			
ѧ	ŭ	ŭ				ă			

<sup>5</sup> The official Bulgarian Romanization system is set by a law since 13 March 2009.