




Pyrodigal: Python bindings and interface to Prodigal, an efficient method for gene prediction in prokaryotes.

Martin Larralde¹

¹ Structural and Computational Biology Unit, EMBL, Heidelberg, Germany

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: 

Submitted: 25 February 2022

Published: unpublished

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

Summary

Improvements in sequencing technologies have seen the amount of available genomic data expand considerably over the last twenty years. One of the key step for analysing is the prediction of protein-coding regions in genomic sequences, known as Open Reading Frames (ORFs), which span between a start and a stop codon. A recent comparison of several ORF prediction methods ([Korandla et al., 2019](#)) has shown that Prodigal ([Hyatt et al., 2010](#)), a prokaryotic gene finder that uses dynamic programming, is one of the highest performing *ab initio* ORF finders. Pyrodigal is a Python package that provides bindings and an interface to Prodigal to make it easier to use in Python applications.

Statement of need

Prodigal is used in thousands of applications as the gene calling method for processing genomic sequences. It is implemented in ANSI C, making it extremely efficient and relatively easy to compile on different platforms. However, the only way to use Prodigal is through an executable, making it inconvenient for bioinformaticians who rely on the increasingly popular Python language. In addition to the hassle caused by the invocation of the executable from Python code, the distribution of Python programs relying on Prodigal is also problematic, since they now require an external binary that cannot be installed from the [Python Package Index](#) (PyPI).

To address these issues, we developed Pyrodigal, a Python module implemented in Cython ([Behnel et al., 2020](#)) that binds to the Prodigal internals, resulting in identical predictions and similar performance through a friendly object-oriented interface. The predicted genes are returned as Python objects, with properties for retrieving confidence scores or coordinates, and methods for translating the gene sequence with a default or user-provided translation table. Prodigal is compiled from source and statically linked into a compiled Python extension, which allows it to be installed with a single `pip install` command, even on a target machine that requires compilation.

Pyrodigal has already been used as the implementation for the initial ORF finding stage in several domains, including biosynthetic gene cluster prediction ([Carroll et al., 2021](#)), prophage identification ([Sirén et al., 2021](#); [Turkington et al., 2021](#)), and pangenome analysis ([Hernández et al., 2021](#)).

Method

Internally, Prodigal identifies start and stop codons throughout a genomic sequence, which are represented as nodes. It then uses a dynamic programming approach to compute scores for every pair of nodes (i.e. putative genes) as shown in [Figure 1](#). Scores assigned to predicted genes are based primarily on the frequency of nucleotide hexamers inside the gene sequence.

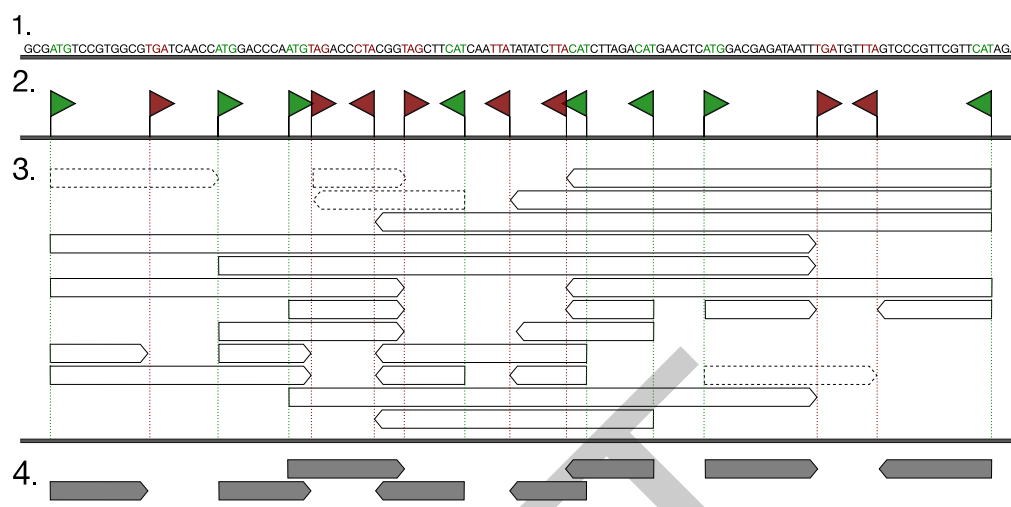


Figure 1: Graphical depiction of the Prodigal method for identifying genes in a sequence. First, the sequence is analysed to find start and stop codons in the 6 reading frames (1). Dynamic programming nodes are then created for each codon, each storing the strand (shown with the triangle direction, right for the direct strand, left for the reverse strand) and the type (green for start, red for stop) of the codon they were obtained from. Nodes are then scored on biological criteria (2). Putative genes are identified between all start and stop codons in a given window (a gene cannot span between any pair of nodes; some invalid connections are shown with dashed lines as examples). Then putative genes are scored using a dynamic programming approach (3). Once all connections have been processed, the dynamic programming matrix is traversed to find the highest scoring path, giving the final predictions (4).

Pyrodigal adapts the first two steps so that the dynamic programming nodes can be extracted directly from a Python string containing the sequence data, rather than requiring formatting to an external file. In addition, the node storage has been reworked to use reallocating buffers, saving memory on smaller sequences. The node and connection scoring steps use the original Prodigal code.

Optimization

Prodigal was profiled with Valgrind (Nethercote & Seward, 2007) to identify critical parts of the original code. Using bacterial genomes from the proGenomes v2.1 database (Mende et al., 2020), we found that for long enough sequences, a large fraction of the CPU cycles was spent in the `score_connection` function.

There are pairs of codons between which a gene can never span, such as two stop codons, or a forward start codon and a reverse stop codon, as shown in Figure 1. Upon inspection, we realized the `score_connection` was still called in invalid cases that could be labelled as such beforehand. Identifying these invalid connections is feasible by checking the strand, type and reading frame of a node pair. Considering two nodes i and j , the connection between them is invalid if any of these boolean equations is true:

- $(T_i \neq STOP) \wedge (T_j \neq STOP) \wedge (S_i = S_j)$
- $(S_i = 1) \wedge (T_i \neq STOP) \wedge (S_j = -1)$
- $(S_i = -1) \wedge (T_i = STOP) \wedge (S_j = 1)$
- $(S_i = -1) \wedge (T_i \neq STOP) \wedge (S_j = 1) \wedge (T_j = STOP)$
- $(S_i = S_j) \wedge (S_i = 1) \wedge (T_i \neq STOP) \wedge (T_j = STOP) \wedge (F_i \neq F_j)$
- $(S_i = S_j) \wedge (S_i = -1) \wedge (T_i = STOP) \wedge (T_j \neq STOP) \wedge (F_i \neq F_j)$

where T_i , S_i and F_i are respectively the type, strand, and reading frame of the node i , with forward and reverse strands encoded as +1 and -1 respectively.

We developed a heuristic filter to quickly identify node pairs forming an invalid connection prior to the scoring step using the above formulas. Since all these attributes have a small number of possible values (+1 or -1 for the forward or reverse strand; *ATG*, *GTG*, *TTG* or *STOP* for the codon type; -1, -2, -3, +1, +2, +3 for the reading frame), they can all be stored in a single byte. Use of the SIMD features of modern CPUs allows several nodes to be processed at once (8 nodes with NEON and SSE2 features, 16 nodes with AVX2). This first pass produces a look-up table used to bypass the scoring of invalid connections.

The performance of the connection scoring was evaluated on 50 bacterial sequences of various length, as shown in Figure 2. It suggests that even with the added cost of the additional pass for each node, enabling the heuristic filter in Pyrodigal saves about half of the time needed to score connections between all the nodes of a sequence.

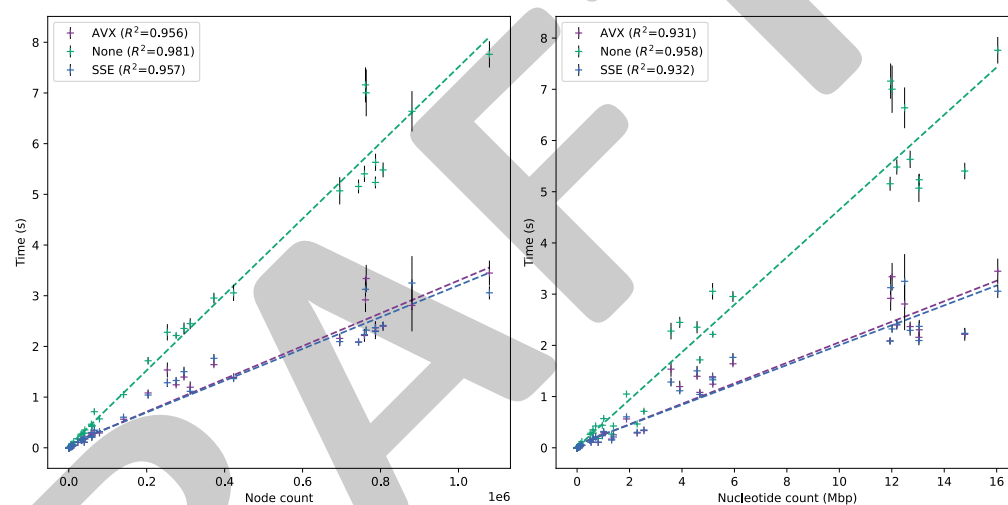


Figure 2: Evaluation of the connection scoring performance with different heuristic filter SIMD backends (SSE2 or AVX2) or without enabling the filter (None). Each sequence was processed 10 times on a quiet i7-8550U CPU @ 1.80GHz.

Availability

Pyrodigal is distributed on PyPI under the [GNU Lesser General Public License](#). Pre-compiled distributions are provided for MacOS, Linux and Windows x86-64, as well as Linux Aarch64 machines. A [Conda](#) package is also available in the Bioconda channel ([Grüning et al., 2018](#)).

The source code is available in a git repository on [GitHub](#), and features a Continuous Integration workflow to run integration tests on changes. Documentation is hosted on [ReadTheDocs](#) and built for each new release.

Acknowledgments

We thank Laura M. Carroll for her input on the redaction of this manuscript, and Georg Zeller for his supervision. This work was funded by the European Molecular Biology Laboratory and the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG, grant no. 395357507 – [SFB 1371](#)).

References

- Behnel, Stefan., Bradshaw, Robert., Dalcín, Lisandro., Florisson, Mark., Makarov, Vitja., Seljebotn, Dag Sverre, & the Cython contributors. (2020). *Cython: C-Extensions for Python*. <https://cython.org>
- Carroll, L. M., Larralde, M., Fleck, J. S., Ponnudurai, R., Milanese, A., Cappio, E., & Zeller, G. (2021). *Accurate de novo identification of biosynthetic gene clusters with GECCO* (p. 2021.05.03.442509) [Preprint]. bioRxiv. <https://doi.org/10.1101/2021.05.03.442509>
- Grüning, B., Dale, R., Sjödin, A., Chapman, B. A., Rowe, J., Tomkins-Tinch, C. H., Valieris, R., & Köster, J. (2018). Bioconda: Sustainable and comprehensive software distribution for the life sciences. *Nature Methods*, 15(7), 475–476. <https://doi.org/10.1038/s41592-018-0046-7>
- Hernández, J. R., Guirado, I. C., Sánchez, M. B., Verdaguer, F. C., & the iGem 2021 ARIA team. (2021). *AlphaMine: An alignment-free genomic analysis system that can build core, shell, and cloud prokaryotic pangenomes by applying set-theory operations to genome collections*. https://2021.igem.org/Team:UPF_Barcelona/Software_AMine
- Hyatt, D., Chen, G.-L., LoCascio, P. F., Land, M. L., Larimer, F. W., & Hauser, L. J. (2010). Prodigal: Prokaryotic gene recognition and translation initiation site identification. *BMC Bioinformatics*, 11, 119. <https://doi.org/10.1186/1471-2105-11-119>
- Korandla, D. R., Wozniak, J. M., Campeau, A., Gonzalez, D. J., & Wright, E. S. (2019). AssessORF: Combining evolutionary conservation and proteomics to assess prokaryotic gene predictions. *Bioinformatics*, 36(4), 1022–1029. <https://doi.org/10.1093/bioinformatics/btz714>
- Mende, D. R., Letunic, I., Maistrenko, O. M., Schmidt, T. S. B., Milanese, A., Paoli, L., Hernández-Plaza, A., Orakov, A. N., Forslund, S. K., Sunagawa, S., Zeller, G., Huerta-Cepas, J., Coelho, L. P., & Bork, P. (2020). proGenomes2: An improved database for accurate and consistent habitat, taxonomic and functional annotations of prokaryotic genomes. *Nucleic Acids Research*, 48(D1), D621–D625. <https://doi.org/10.1093/nar/gkz1002>
- Nethercote, N., & Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 89–100. <https://doi.org/10.1145/1250734.1250746>
- Sirén, K., Millard, A., Petersen, B., Gilbert, M. T. P., Clokie, M. R. J., & Sicheritz-Pontén, T. (2021). Rapid discovery of novel prophages using biological feature engineering and machine learning. *NAR Genomics and Bioinformatics*, 3(1), lqaa109. <https://doi.org/10.1093/nargab/lqaa109>
- Turkington, C. J. R., Abadi, N. N., Edwards, R. A., & Grasis, J. A. (2021). *hafeZ: Active prophage identification through read mapping* [Preprint]. Bioinformatics. <https://doi.org/10.1101/2021.07.21.453177>