The Journal of Open Source Software

pyscreener: A Python Wrapper for Computational Docking Software

David E. Graff [12] **and Connor W. Coley** [23¶]

1 Department of Chemistry and Chemical Biology, Harvard University **2** Department of Chemical Engineering, Massachusetts Institute of Technology **3** Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology ¶ Corresponding author

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Richard Gowers

Reviewers:

- @mikemhenry
- @rvhonorato

Submitted: 17 November 2021
Published: unpublished

Summary

pyscreener is a Python library that seeks to alleviate the challenges of large-scale structure-based design using computational docking. It provides a simple and uniform interface that is agnostic to the backend docking engine with which to calculate the docking score of a given molecule in a specified active site. Additionally, pyscreener features first-class support for task distribution, allowing users to seamlessly scale their code from a local, multi-core setup to a large, heterogeneous resource allocation.

Statement of Need

Computational docking is an important technique in structure-based drug design that enables the rapid approximation of binding affinity for a candidate ligand in a matter of CPU seconds. With the growing popularity of ultra-large ligand libraries, docking is increasingly used to sift through hundreds of millions of compounds to try to identify novel and potent binders for a variety of protein targets (Gorgulla et al., 2020; Lyu et al., 2019). There are many choices of docking software, and certain software are better suited towards specific protein-ligand contexts (e.g., flexible protein side chains or sugar-like ligand molecules). Switching between these software is often not trivial as the input preparation, simulation, and output parsing pipelines differ between each software.

In addition, many of these programs exist only as command-line applications and lack Python bindings. This presents an additional challenge for their integration into molecular optimization workflows, such as reinforcement learning or genetic algorithms. Molecular optimization objectives have largely been limited to benchmark tasks, such as penalized logP, QED, JNK3 or GSK3β inhibitor classification (Li et al., 2018), and others contained in the GuacaMol library (Brown et al., 2019). These benchmarks are useful for comparing molecular design techniques, but they are not representative of true drug discovery tasks in terms of complexity; computational docking is at least one step in the right direction.

While many molecular optimization techniques propose new molecules in the form of SMILES strings (Elton et al., 2019), most docking programs accept input in the form of molecular supply files with predefined 3D geometry (e.g., Mol2 or PDBQT format). Using the docking score of a molecule as a design objective thus requires an ad hoc implementation for which no standardized approach exists. The `vina` library (Eberhardt et al., 2021) is currently the only library capable of performing molecular docking within Python code, but it is limited to docking molecules using solely AutoDock Vina as the backend docking engine. Moreover, the object model of the `vina` library accepts input ligands only as PDBQT files or strings and still does not address the need to quickly calculate the docking score of a molecule from its SMILES string.

In our work on the MolPAL software (Graff et al., 2021), we required a library that is able to accept molecular inputs as SMILES strings and output their corresponding docking scores for a given receptor and docking box. Our use-case also called for docking large batches of molecules across large and distributed hardware setups. Lastly, we desired that our library be flexible with respect to the underlying docking engine, allowing us to use a variety of backend docking software (e.g., Vina (Trott & Olson, 2010), Smina (Koes et al., 2013), QVina (Alhossary et al., 2015), or DOCK6 (Allen et al., 2015)) with minimal changes to client code. To that end, we developed `pyscreener`, a Python library that is flexible with respect to both molecular input format and docking engine that transparently handles the distribution of docking simulations across large resource allocations.

Implementation and Performance

The primary design goals with `pyscreener` were to (1) provide a simple interface with which to calculate the docking score of an input small molecule and (2) transparently distribute the corresponding docking simulations across a large resource allocation. The object model of `pyscreener` relies on four classes: `CalculationData`, `CalculationMetadata`, `DockingRunner`, and `DockingVirtualScreen`. A docking simulation in `pyscreener` is fully described by a `CalculationData` and an associated `CalculationMetadata`. High-level information about the simulation that is common to all docking software (e.g., target protein, docking box, name of the ligand, the paths under which inputs and outputs will be stored) is stored in the `CalculationData` object. A `CalculationMetadata` contains the set of software-specific arguments for the simulation, such as `exhaustiveness` for the AutoDock Vina family of software or parameters for SPH file preparation for DOCK6.

The `pyscreener` object model separates data from behavior by placing the responsibility of actually preparing, running, and parsing simulations inside the `DockingRunner` class. This stateless class defines methods to prepare simulation inputs, perform the simulation of the corresponding inputs, and parse the resulting output for a given `CalculationData` and `CalculationMetadata` pair. By placing this logic inside static methods rather than attaching them to the `CalculationData` object, `pyscreener` limits network data transfer overhead during task distribution. The separation also allows for the straightforward addition of new backend docking engines to `pyscreener`, as this entails only the specification of the corresponding `CalculationMetadata` and `DockingRunner` subclasses.

`pyscreener` also contains the `DockingVirtualScreen` class, which contains a template `CalculationData` and `CalculationMetadata` with which to dock each input molecule, i.e., a virtual screening protocol. The class defines a `__call__()` method which takes SMILES strings or chemical files in any format supported by OpenBabel (O'Boyle et al., 2011) as input and distributes the corresponding docking simulations across the resources in the given hardware allocation, returning a docking score for each input molecule.

The Journal of Open Source Software

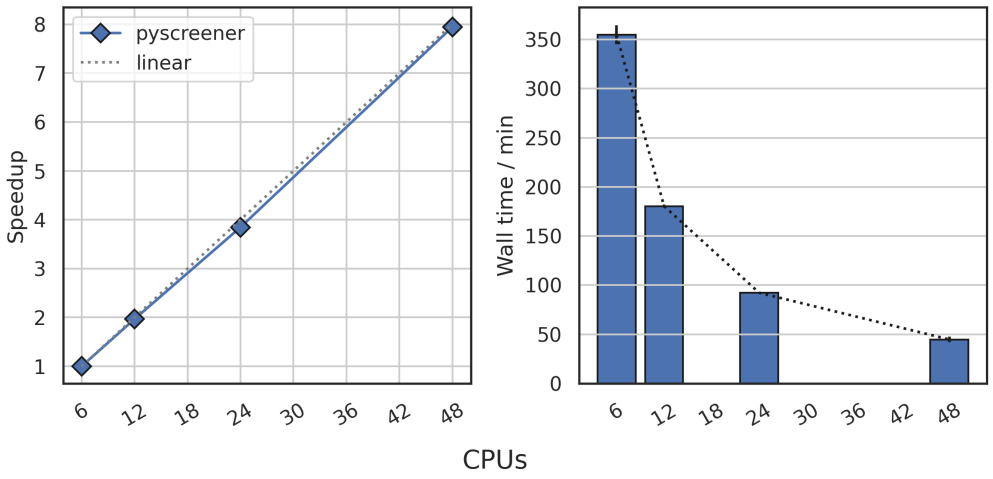

Figure 1: Wall-time of the computational docking of all 1,615 FDA-approved drugs against 5WIU using QVina2 over six CPU cores for a single-node setup with the specified number of CPU cores. (Left) calculated speedup. (Right) wall time in minutes. Bars reflect mean ± standard deviation over three runs.

To handle task distribution, `pyscreener` relies on the `ray` library (Moritz et al., 2018) for distributed computation. For multithreaded docking software, `pyscreener` allows a user to specify how many CPU cores to run each individual docking simulation over, running as many docking simulations in parallel as possible for a given number of total CPU cores in the `ray` cluster. To examine the scaling behavior of `pyscreener`, we docked all 1,615 FDA-approved drugs into the active site of the D4 dopamine receptor (PDB ID 5WIU (Wang et al., 2017)) with QVina2 running over 6 CPU cores. We tested both single node hardware setups, scaling the total number of CPU cores on one machine, and multi-node setups, scaling the total number of machines. In the single-node case, `pyscreener` exhibited essentially perfect scaling Figure 1 as we scaled the size of the `ray` cluster from 6 to 48 CPU cores running QVina over 6 CPU cores.

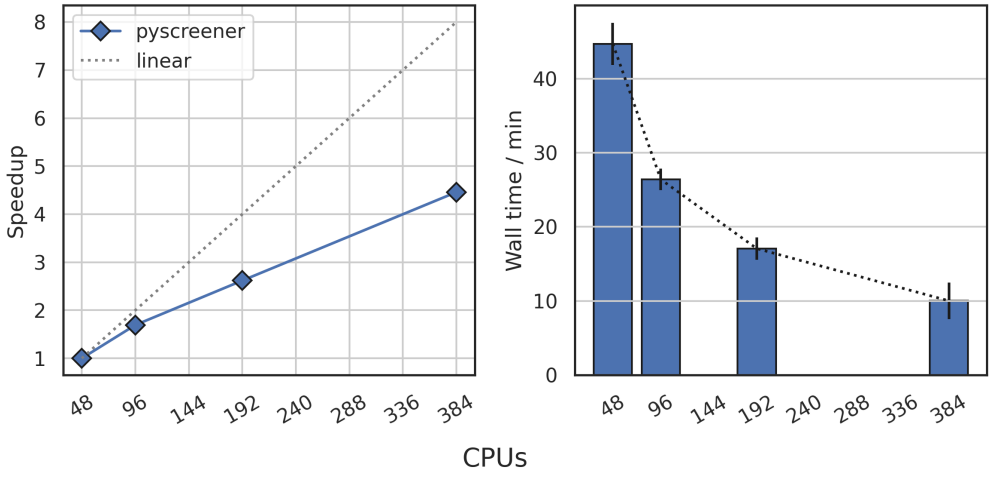

Figure 2: Wall-time of the computational docking of all 1,615 FDA-approved drugs against 5WIU using QVina2 over six CPU cores for setups using multiple 48-core nodes with the total number of specified CPU cores. (Left) calculated speedup. (Right) wall time in minutes. Bars reflect mean ± standard deviation over three runs.

In contrast, the multi-node setup exhibits less ideal scaling Figure 2 with a measured speedup approximately 55% that of perfect scaling. We attribute this scaling behavior to hardware-dependent network communication overhead. Distributing a `sleep(5)` function allocated 6 CPU cores per task (to mimic a fairly quick docking simulation) in parallel over differing hardware setups led to an approximate 2.5% increase in wall-time relative to the single-node setup each time the number of nodes in the setup was doubled while keeping the total number of CPU cores the same. Such a trend is consistent with network communication being detrimental to scaling behavior. This test also communicated the absolute minimum amount of data over the network, as there were no function arguments or return values. When communicating `CalculationData` objects (approximately 600 bytes in serialized form) over the network, as in `pyscreener`, the drop increased to 6% for each doubling of the total number of nodes. Minimizing the total size of `CalculationData` objects was therefore an explicit implementation goal. Future development will seek to further reduce network communication overhead costs to bring `pyscreener` scaling closer to ideal scaling.

Examples

To illustrate `pyscreener`, we consider docking benezene (SMILES string `"c1ccccc1"`) against 5WIU with a docking box centered at (-18.2, 14.4, -16.1) with x-, y-, and z-radii (15.4, 13.9, 14.5). We may perform this docking using AutoDock Vina over 6 CPU cores via `pyscreener` like so:

```
>>> import pyscreener as ps
>>> metadata = ps.build_metadata("vina")
>>> virtual_screen = ps.virtual_screen(
...    "vina", receptors=["5WIU.pdb"],
...    center=(-18.2, 14.4, -16.1),
...    size=(15.4, 13.9, 14.5),
...    metadata_template=metadata,
...    ncpu=6
... )
>>> scores = virtual_screen("c1ccccc1")
>>> scores
array([-4.4])
```

Alternatively, we may dock many molecules by passing a `List` of SMILES strings to the `DockingVirtualScreen`:

```
>>> smis = [
...    "c1ccccc1",
...    "O=C(Cc1ccccc1)NC1C(=O)N2C1SC(C2C(=O)O)(C)C",
...    "C=CCN1CCC23C4C(=O)CCC2(C1CC5=C3C(=C(C=C5)O)O4)O"
... ]
>>> scores = virtual_screen(smis)
>>> scores.shape
(3,)
```

By default, AutoDock Vina docks molecules using an `--exhaustiveness` value of 8, but we may specify a higher number in the `metadata`:

```
>>> metadata = ps.build_metadata("vina", dict(exhaustivness=32))
```

We may also utilize other docking engines in the AutoDock Vina family by specifying the `software` for Vina-type metadata. Here, we use the accelerated optimization routine of QVina for faster docking. Note that we also support `software` values of `"smina"` (Koes et al., 2013) and `"psovina"` (Ng et al., 2015) in addition to `"vina"` (Trott & Olson, 2010) and `"qvina"` (Alhossary et al., 2015).

```
>>> metadata = ps.build_metadata("vina", dict(software="qvina"))
```

It is also possible to dock molecules using DOCK6 (Allen et al., 2015) in pyscreener. To do this, we must first construct DOCK6 metadata and specify that we are creating a DOCK6 virtual screen (note that DOCK6 is not multithreaded and thus does not benefit from being assigned multiple CPU cores per task):

```
>>> metadata = ps.build_metadata("dock")
>>> virtual_screen = ps.virtual_screen(
...   "dock",
...   receptors=["5WIU.pdb"],
...   center=(-18.2, 14.4, -16.1),
...   size=(15.4, 13.9, 14.5),
...   metadata_template=metadata
... )
>>> scores = virtual_screen("c1ccccc1")
>>> scores
array([-12.35])
```

Acknowledgements

The authors thank Keir Adams and Wenhao Gao for providing feedback on the preparation of this paper and the pyscreener code. The computations in this paper were run on the FASRC Cannon cluster supported by the FAS Division of Science Research Computing Group at Harvard University. The authors also acknowledge the MIT SuperCloud and Lincoln Laboratory Supercomputing Center for providing HPC and consultation resources that have contributed to the research results reported within this paper. This work was funded by the MIT-IBM Watson AI Lab.

References

Alhossary, A., Handoko, S. D., Mu, Y., & Kwoh, C.-K. (2015). Fast, accurate, and reliable molecular docking with QuickVina 2. *Bioinformatics*, *31*(13), 2214–2216. https://doi.org/10.1093/bioinformatics/btv082

Allen, W. J., Balius, T. E., Mukherjee, S., Brozell, S. R., Moustakas, D. T., Lang, P. T., Case, D. A., Kuntz, I. D., & Rizzo, R. C. (2015). DOCK 6: Impact of new features and current docking performance. *Journal of Computational Chemistry*, *36*(15), 1132–1156. https://doi.org/10.1002/jcc.23905

Brown, N., Fiscato, M., Segler, M. H. S., & Vaucher, A. C. (2019). GuacaMol: Benchmarking Models for de Novo Molecular Design. *Journal of Chemical Information and Modeling*, *59*(3), 1096–1108. https://doi.org/10.1021/acs.jcim.8b00839

Eberhardt, J., Santos-Martins, D., Tillack, A. F., & Forli, S. (2021). AutoDock Vina 1.2.0: New Docking Methods, Expanded Force Field, and Python Bindings. *Journal of Chemical Information and Modeling*, *61*(8), 3891–3898. https://doi.org/10.1021/acs.jcim.1c00203

Elton, D. C., Boukouvalas, Z., Fuge, M. D., & Chung, P. W. (2019). Deep learning for molecular design—a review of the state of the art. *Molecular Systems Design & Engineering*, *4*(4), 828–849. https://doi.org/10.1039/C9ME00039A

Gorgulla, C., Boeszoermenyi, A., Wang, Z.-F., Fischer, P. D., Coote, P. W., Padmanabha Das, K. M., Malets, Y. S., Radchenko, D. S., Moroz, Y. S., Scott, D. A., Fackeldey, K., Hoffmann, M., Iavniuk, I., Wagner, G., & Arthanari, H. (2020). An open-source drug discovery platform enables ultra-large virtual screens. *Nature*, *580*(7805), 663–668. https://doi.org/10.1038/s41586-020-2117-z

Graff, D. E., Shakhnovich, E. I., & Coley, C. W. (2021). Accelerating high-throughput virtual screening through molecular pool-based active learning. *Chemical Science*, *12*(22), 7866–7881. https://doi.org/10.1039/D0SC06805E

Koes, D. R., Baumgartner, M. P., & Camacho, C. J. (2013). Lessons Learned in Empirical Scoring with smina from the CSAR 2011 Benchmarking Exercise. *Journal of Chemical Information and Modeling*, *53*(8), 1893–1904. https://doi.org/10.1021/ci300604z

Li, Y., Zhang, L., & Liu, Z. (2018). Multi-objective de novo drug design with conditional graph generative model. *Journal of Cheminformatics*, *10*(1), 33. https://doi.org/10.1186/s13321-018-0287-6

Lyu, J., Wang, S., Balius, T. E., Singh, I., Levit, A., Moroz, Y. S., O'Meara, M. J., Che, T., Algaa, E., Tolmachova, K., Tolmachev, A. A., Shoichet, B. K., Roth, B. L., & Irwin, J. J. (2019). Ultra-large library docking for discovering new chemotypes. *Nature*, *566*(7743), 224–229. https://doi.org/10.1038/s41586-019-0917-9

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., & Stoica, I. (2018). Ray: A Distributed Framework for Emerging AI Applications. *arXiv:1712.05889 [Cs, Stat]*. http://arxiv.org/abs/1712.05889

Ng, M. C. K., Fong, S., & Siu, S. W. I. (2015). PSOVina: The hybrid particle swarm optimization algorithm for protein–ligand docking. *Journal of Bioinformatics and Computational Biology*, *13*(03), 1541007. https://doi.org/10.1142/S0219720015410073

O'Boyle, N. M., Banck, M., James, C. A., Morley, C., Vandermeersch, T., & Hutchison, G. R. (2011). Open Babel: An open chemical toolbox. *Journal of Cheminformatics*, *3*(1), 33. https://doi.org/10.1186/1758-2946-3-33

Trott, O., & Olson, A. J. (2010). AutoDock Vina: Improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of Computational Chemistry*, *31*(2), 455–461. https://doi.org/10.1002/jcc.21334

Wang, S., Wacker, D., Levit, A., Che, T., Betz, R. M., McCorvy, J. D., Venkatakrishnan, A. J., Huang, X.-P., Dror, R. O., Shoichet, B. K., & Roth, B. L. (2017). D4 dopamine receptor high-resolution structures enable the discovery of selective agonists. *Science*, *358*(6361), 381–386. https://doi.org/10.1126/science.aan5468