The Journal of Open Source Software

Raster Tools: An Open Source Toolbox for Raster Processing

Fredrick Bunt [1], **Jesse Johnson** [1], **and John Hogland** [2]

1 University of Montana, USA **2** Rocky Mountain Research Station, USA

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Kanishka B. Narayan

Reviewers:

- @nagellette
- @kanishkan91

Submitted: 10 January 2025
Published: unpublished

Summary

The Raster Tools Python package provides an application programmers interface (API) for building raster data processing pipelines. Pipelines built with Raster Tools can efficiently process extremely large raster datasets, including those larger than the system's available memory. Raster Tools can be deployed on systems ranging from small laptops to high performance computing environments. To aid in the construction of raster processing pipelines, Raster Tools provides a consistent API to a suite of scalable and lazy raster processing functions including focal, zonal, clipping, convolution, and distance analysis operations. Raster Tools makes it much easier and more accessible to perform research using today's massive raster datasets.

Statement of Need

Python has become very popular as a language for data processing. This is because it provides a rich, general purpose, data stack, and simple, easy to understand syntax. Because of this, it is a good choice for tackling the expanding data problem. Some key libraries in the stack are Numpy (Harris et al., 2020), Xarray (Hoyer & Hamman, 2017), SciPy (Virtanen et al., 2020), and Dask (Dask Development Team, 2016). Numpy forms the foundation of Python's stack by providing a fast and powerful N-dimensional (ND) array implementation and API for manipulating them. The Xarray package builds on this by supplying an API for working with labelled ND datasets. This is especially relevant to raster data with labeled dimensions and coordinates. Its data structures can be backed by any array object that implements Numpy's array interface, with Numpy being the default. The SciPy package provides optimized implementations of a wide variety of scientific and engineering algorithms.

The Python data stack also includes packages for working with geospatial data. The two main packages are both Python bindings for GDAL (GDAL/OGR contributors, 2024). The first is the GDAL project's package of the same name. This package is a thin wrapper around the C/C++ library and can be cumbersome and difficult to use. The second is Rasterio (Gillies & others, 2013--), which provides a more modern and "Pythonic" API for working with geospatial data, but does not implement many of GDAL's features. A third package, rioxarray (rioxarray Development Team, 2019), is a plugin for Xarray that allows raster formats to be easily read into Xarray's data structures and for those data structures to use Rasterio operations on the underlying data. Finally, odc-geo (OpenDataCube, 2025) provides several geospatial tools and algorithms such as grid definitions and reprojection. It also can function as a plugin for Xarray.

Together, these packages and the wider Python data stack can be leveraged to carry out simple and advanced raster processing and geographic information systems (GIS) work. There are some limitations, however. Several merit consideration. First, in the GIS world, null or missing data is typically handled with sentinel values that are outside of the data domain (e.g. -9999), while in much of Python's data stack, this is handled with NaN values. These two paradigms

can be difficult to reconcile, and it requires boilerplate code to properly mask data. Second, Numpy does not support parallel processing without writing C extensions, which is difficult at best. Newer packages such as Numba (Lam et al., 2015) and Numexpr (McLeod, 2018) can help with this, but their scope is still limited. They also require more boilerplate code to be written. Third, if the size of the target dataset approaches or exceeds the available memory, Numpy and everything built on it will exhaust the available memory and fail. This is because Numpy and most of the Python data stack is "in-core." All the data is in memory when processing occurs. To tackle this problem, more boilerplate code must be written to manually chunk the data and process it in batches.

The last two issues can be solved by using the Dask package. Dask provides a relatively simple interface for parallel computing in Python. It allows large arrays to be chunked into smaller pieces and processed in parallel using its schedulers. This allows for out-of-core (OOC) or external memory processing of datasets, where only part of the dataset is loaded into memory at any given time. Dask implements much of Numpy's API and can therefore be used to replace Numpy arrays in data pipelines, with some boilerplate applied. Xarray and odc-geo have first class support for Dask, while rioxarray has partial support for it.

With the packages described above, it is possible to perform both simple and advanced data and GIS processing on large datasets. It can be time consuming and difficult, however, to implement. Boilerplate code is needed to cover compatibility gaps, such as making Rasterio and SciPy functions compatible with Dask. Code is also needed to properly handle missing data and to implement common and advanced GIS operations. There is no package that does all of this for the user.

Current State of the Field

Several open-source Python packages try to address these needs in the Python ecosystem, but each does so differently and has its own limitations. Some examples are WhteboxTools (J. Lindsay, 2018; J. B. Lindsay, 2016), GeoUtils (G. developers, 2020--2026), and Xarray-Spatial (X.-S. developers, 2020--2026).

WhiteboxTools is a geospatial library providing hundreds of raster and GIS analysis functions through Python bindings. It is compiled and highly performant and capable of processing raster data in parallel. However, it presents several limitations: it lacks integration with the Python data stack and requires data to be fully loaded into memory for processing. Additionally, operations cannot be chained in memory; each tool must read from storage, process the data, and write the output back to storage before the next step can begin. A newer adaptation, Whitebox Workflows for Python, has been released, which allows chaining. It is proprietary freeware with a paid upgrade package, however.

Built on top of Rasterio, GeoUtils is a package designed for efficient geospatial analysis with Python operator overloading and NumPy integration. It leverages Python's multiprocessing module to process rasters in parallel. However, the package has notable limitations: most operations are eager and load the full dataset into memory, and it only implements a few geospatial operations such as reproject and proximity. Additionally, GeoUtils currently lacks compatibility with Dask and offers limited integration with Xarray. Work is ongoing to add Dask support, and an Xarray extension is planned. xDEM is a package from the same authors, built on top of GeoUtils, which provides a suite of digital elevation model (DEM) focused functions.

Finally, Xarray-Spatial is a package designed for spatial analysis of rasters, built directly on Xarray data objects. It provides a wide variety of analysis functions and features acceleration via Numba and scalability through Dask support. Additionally, the library offers GPU capabilities by leveraging CuPy (Okuta et al., 2017) and Numba's just-in-time (JIT) cuda compilation. However, the package has some limitations, including an inflexible and, sometimes, inconsistent interface where support for Dask, GPU acceleration, and parallel processing varies across

different functions. Data type support is also restricted, with most functions converting inputs to float 32 and using NaNs for missing data, and the library currently lacks some common raster operations found in other tools.

Software Description

We introduce the Raster Tools package as a solution to the above problems and as a tool to build pipelines for processing of both large and small-scale raster datasets. Raster Tools seeks to bridge the gaps in the Python data stack by taking care of the necessary boilerplate. It also provides a platform for carrying out GIS processing and analysis by implementing a suite of common and advanced GIS operations that are OOC compatible. These operations would normally be pulled from several other packages, implemented from scratch, or some combination of the two. By taking care of these tasks, Raster Tools enables researchers to tackle large problems and datasets whose size may have been prohibitive before.

At the core of Raster Tools is the Raster class. This class provides several useful features that help with raster processing such as operator overloading, Numpy universal function (Ufunc) and aggregation compatibility, and the ability to open or save to any GDAL supported format. With operator overloading, Raster objects behave much like Numpy arrays and can be used with any of Python's arithmetic, logical, bitwise, and comparison operators. By implementing Numpy's Ufunc interface, Raster objects provide compatibility with Numpy's element-wise functions such as sin, cos, sqrt, exp, etc. The result of calling a Numpy Ufunc on a Raster object will always be another Raster object. These two features, together, make implementing mathematical models as easy as writing the equations as expressions in code. All these operations are missing-data aware. The Raster class implements a lazy evaluation model inherited from Dask. Computational graphs are constructed during operation chaining, deferring data loading and processing until an explicit trigger, such as saving to disk or explicit loading, is executed. This enables the definition of complex pipelines that exceed local memory limits. The code listing below shows an example of an index being calculated from Landsat 8 bands and saved to disk.

```
import numpy as np
import raster_tools as rts

# Create two Raster objects pointing to the red and near-IR bands
red = rts.Raster("data/landsat_stack.tif").get_bands(4)
nir = rts.Raster("data/landsat_stack.tif").get_bands(5)
# Calculate the MSAVI2 index. This is lazy and no computation has occurred.
msavi2 = (2 * (nir + 1) - np.sqrt((2 * nir + 1) ** 2 - 8 * (nir - red))) / 2
# Save to storage. This starts computation.
msavi2.save( "outputs/msavi2.tif", tiled=True, bigtiff="IF_NEEDED", compress="LZW" )
```

In conjunction with the Raster class, Raster Tools provides an API for raster processing. This API consists of the following modules:

- `clipping`: functions for clipping rasters using vector geometries.
- `creation`: functions for creating rasters based off a reference grid with analogs to Numpy's ones_like, zeros_like, etc, as well as for generating data from random distributions.
- `distance`: functions for computing proximity and cost-distance analysis.
- `focal`: functions for applying focal operations,including convolution, to rasters.
- `general`: general purpose functions such as raster value remapping.
- `line_stats`: advanced function for calculating statistics on line-vectors overlayed on a raster.
- `rasterize`: functions for rasterizing vector geometries.
- `surface`: functions for transforming DEMs, such as aspect, slope, and 3D surface area.
- `vector`: functions for handling vector data.

- `warp`: functions for reprojecting rasters.
- `zonal`: functions for computing zonal statistics.

All parts of the API are lazy and parallel with one exception. The cost-distance analysis functions are fully in memory and non-parallel due to the difficulty in parallelizing the algorithm. The results returned by the cost-distance analysis functions are Raster objects and thus still compatible with the rest of the API. With the suite of operations provided in the API, it is possible to carry out a wide range of raster analysis and processing tasks.

Because the API is lazy, it is possible to chain many operations together before saving results back to disk. This can prevent the need to write an intermediate product after each operation and greatly reduce read/write overhead in processing pipelines as well as storage needs.

Compatibility

Raster Tools offers a high degree of compatibility with the wider Python data stack. The Raster class has methods and attributes for converting raster objects into Numpy and Dask arrays, Xarray DataArray and Dataset objects, and dask-geopandas GeoDataFrame vectors (Bossche et al., 2025). This compatibility works in the opposite direction as well. The Raster Tools classes and functions can ingest Numpy and Dask arrays and Xarray objects. This extends to the overloaded operators. It is possible to multiply a Numpy array against a Raster object with matching spatial dimensions and get a new Raster with the expected result. Because of Raster Tools' compatibility, it is possible to easily convert data to an appropriate format for another library and then ingest results back into Raster Tools, or vice versa.

Raster Tools also provides the Model Predict API. This API allows a model to be applied to any raster in a lazy fashion, compatible with the rest of the package. Using the Model Predict API allows models not normally compatible with Dask or Raster Tools to be easily folded into pipelines.

Scaling

To evaluate the performance and scaling of Raster Tools, we benchmarked it in a controlled computing environment. The environment used was a virtual machine with 400 GiB of memory and a variable number of CPU cores. We created several scenarios for benchmarking. To test scaling, we set the CPU count at 2, 4, 8, 16, and 32 and ran all the scenarios for each count. The scenarios are described in the table below. The input rasters for the scenarios, labeled A, B, and C below, were copies of the same 30m digital elevation model (DEM) of the continental US (CONUS). These were single precision (float 32) rasters and roughly 60 GB uncompressed on disk. The timing for the scenarios was done using Python's `time` module and includes both loading of the input rasters and writing of the result rasters. A chunk size of 256 MiB was used. Testing was performed in Q3-Q4 2025. At the time, Raster Tools was only compatible with Dask's default scheduler, so that was the Dask scheduler used. The results shown here should be taken as a baseline, since the new scheduler is expected to improve performance.

Scenario	Basic Forumula	Description

Scenario	Basic Forumula	Description
Raster Math	A + B + C	Adding three rasters together
Transcendental Functions	sin(A) * cos(B) / exp(C)	Combining three transcendental functions
Focal Mean	focal_stats(A, “mean”, 5)	Calculate the focal mean with a 5x5 window
Focal Entropy	focal_stats(A, “entropy”, 5)	Calculate the focal entropy with a 5x5 window
Slope	slope(A)	Calculate the slope of a DEM
Curvature	curvature(A)	Calculate the curvature of the DEM
Reproject	reproject(A, new_crs)	Calculate the curvature of the DEM

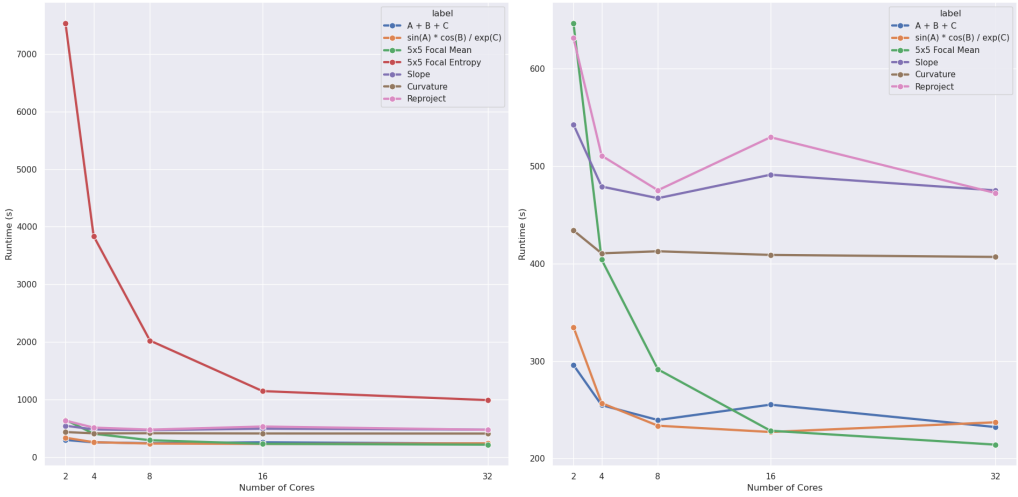

Figure 1: Left) Runtimes for all scenarios. **Right**) The runtimes for all scenarios, excluding the Focal Entropy scenario.

Figure 1 shows the runtimes of the scenarios as the number of cores being used increases. Filtering out the Focal Entropy case for the right-hand plot allows the results for the other scenarios to be viewed more clearly. The Focal Entropy scenario shows strong scaling with the number of cores. Focal Mean shows moderate scaling. The rest show scaling that varies from none to mild.

Use in Research

Raster Tools has been used in several research projects. It was used in Johnson et al. (2022) to process 35 years of Monitoring Trends in Burn Severity (MTBS) fire severity raster data for all of Montana into a tabular format for model training. It was used in Lahrichi et al. (2025) to process burn mask rasters for use in training fire spread prediction models. Raster tools was also used to process rasters, rasterize vectors, and complete a wide range of geospatial analysis in Wiard (2025) and Wiard-Greene et al. (2026).

Raster Tools is also being used in production by the Forest Service for their Potential Control Locations (PCL) product (Connor et al., 2017), (O'Connor et al., 2022). When initially created,

the PCL product was a static, yearly raster for the western half of the United States at 30m resolution. The pipeline for producing PCL was large and time intensive. In 2023 and 2024, the pipeline was completely rebuilt using Raster Tools for its raster processing. The current PCL product is now dynamic and can be produced hourly for local fire support.

Limitations

Raster Tools has limitations which we acknowledge here. During the benchmarking period (Q3–Q4 2025), Raster Tools was constrained to Dask's default threaded scheduler due to a dependency being incompatible with the newer Distributed scheduler (Dask developers, 2026). While this incompatibility has since been resolved, the performance data in Figure 1 reflects the overhead inherent to the threaded scheduler. As shown, scheduling overhead dominates the runtime for lighter tasks. However, scenarios involving compute-intensive operations (high computation-to-chunk-size ratio) essentially mask this overhead, demonstrating that the package scales well when the scheduler is not the bottleneck.

Therefore, the presented benchmarks represent a performance baseline. Since the Distributed scheduler is generally more performant and handles memory backpressure more effectively, users can expect performance to match or exceed these figures.

A related limitation of the threaded scheduler is its prioritization of CPU utilization over memory constraints. Memory consumption is proportional to the number of cores multiplied by the data-chunk size. On systems with a high-core-count, this requires significant available memory. While reducing Dask's target chunk size mitigates this, it increases the number of chunks and adds scheduling overhead.

Importantly, Raster Tools is designed to be scheduler and configuration agnostic. It builds the computational graph but does not enforce a specific execution engine or set of configurations. This ensures that users retain full control over Dask configuration, including the ability to switch to the Distributed scheduler. It also means that Raster Tools will automatically benefit from future improvements in the Dask ecosystem.

Comparison

Except for cost-distance functions, Raster Tools provides a lazy, parallel, OOC, flexible, consistent, and chainable API, along with a suite of raster processing functions, and fully integrates with the wider Python data stack. This contrasts with other packages in the field like those listed above, which force a trade-off in one way or another, between these traits.

Conclusion

Raster Tools fills the gaps in the Python ecosystem by simplifying raster and GIS workflows and reducing the need for boilerplate code. Its compatibility with widely used libraries and support for lazy, OOC operations make it easier for researchers to process large and complex datasets without reinventing common geospatial functions. By providing a unified API for raster analysis, Raster Tools accelerates the development of data pipelines and opens new opportunities for research projects that previously faced technical or resource limitations. Raster Tools has already proven valuable in real-world studies and operational contexts, and it continues to expand the possibilities for geospatial science and applied research.

Bossche, J. V. den, Fleischmann, M., Statham, T., Augspurger, T., (dahn), D. J., Signell, J., Lusk, D., Bunt, F., Gadomski, P., Hagen, R., Bell, R., Lumnitz, S., bernardpazio, RichardScottOZ, Morris, M., Miclat, J., Baker, J., Bourbeau, J., Truong, I., … Zaidi, A. A. (2025). *Geopandas/dask-geopandas: v0.5.0* (Version v0.5.0). Zenodo. https://doi.org/10.5281/zenodo.15579702

Connor, C. D. O., Calkin, D. E., & Thompson, M. P. (2017). An empirical machine learning method for predicting potential fire control locations for pre-fire planning and operational fire management. *International Journal of Wildland Fire*, *26*(7), 587–597. https://doi.org/10.1071/wf16135

Dask developers. (2026). *Dask distributed documentation*. https://distributed.dask.org/

Dask Development Team. (2016). *Dask: Library for dynamic task scheduling*. http://dask.pydata.org

developers, G. (2020--2026). *GeoUtils: Consistent geospatial analysis in python*. https://github.com/GlacioHack/geoutils

developers, X.-S. (2020--2026). *Xarray-spatial*. https://github.com/xarray-contrib/xarray-spatial

GDAL/OGR contributors. (2024). *GDAL/OGR geospatial data abstraction software library*. Open Source Geospatial Foundation. https://doi.org/10.5281/zenodo.5884351

Gillies, S., & others. (2013--). *Rasterio: Geospatial raster i/o for Python programmers*. Mapbox. https://github.com/rasterio/rasterio

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hoyer, S., & Hamman, J. (2017). Xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, *5*(1). https://doi.org/10.5334/jors.148

Johnson, J., Marcozzi, A., Bunt, F., Bova, J., & Hogland, J. (2022). Predicting fire severity in montana using a random forest classification scheme. In D. X. Viegas & L. M. Ribeiro (Eds.), *Advances in fire research 2022* (pp. 323–328). University of Coimbra Press. https://doi.org/10.14195/978-989-26-2298-9_51

Lahrichi, S., Bova, J., Johnson, J., & Malof, J. (2025). *Improved wildfire spread prediction with time-series data and the WSTS+ benchmark*. https://arxiv.org/abs/2502.12003

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A llvm-based python jit compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 1–6.

Lindsay, J. (2018). *WhiteboxTools user manual*. https://www.whiteboxgeo.com/manual/wbt_book/preface.html

Lindsay, J. B. (2016). Whitebox GAT: A case study in geomorphometric analysis. *Computers & Geosciences*, *95*, 75–84. http://dx.doi.org/10.1016/j.cageo.2016.07.003

McLeod, R. (2018). *NumExpr: Fast numerical expression evaluator for NumPy*. https://doi.org/10.5281/zenodo.2483274

O'Connor, C. D., Haas, J. R., Gannon, B. M., Dunn, C. J., Thompson, M. P., & Calkin, D. E. (2022). Modelling potential control locations: Development and adoption of data-driven analytics to support strategic and tactical wildfire containment decisions. *Environmental Sciences Proceedings*, *17*(1). https://doi.org/10.3390/environsciproc2022017073

Okuta, R., Unno, Y., Nishino, D., Hido, S., & Loomis, C. (2017). CuPy: A NumPy-compatible library for NVIDIA GPU calculations. *Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Thirty-First Annual Conference on Neural Information Processing Systems (NIPS)*. http://learningsys.org/nips17/assets/papers/paper_16.pdf

OpenDataCube. (2025). *Odc-geo*. https://github.com/opendatacube/odc-geo

rioxarray Development Team. (2019). *Rioxarray: Geospatial xarray extension powered by*

rasterio. Corteva, Inc. https://github.com/corteva/rioxarray

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Wiard, L. A. (2025). *Investigating the impact of aerial firefighting on rate of wildfire spread* [Master's thesis, University of Montana; University of Montana]. https://scholarworks.umt.edu/etd/12531

Wiard-Greene, L., Johnson, J., Hogland, J., Bunt, F., & Bova, J. (2026). Investigating the impact of aerial firefighting on rate of wildfire spread. *Fire*, *9*(1). https://doi.org/10.3390/fire9010002