The Journal of Open Source Software

quicfire-tools: A Python package for QUIC-Fire simulation management

Anthony A. Marcozzi [1], **Niko J. Tutland** [1], **and Zachary Cope** [2]

1 New Mexico Consortium, Center for Applied Fire and Ecosystem Sciences, Los Alamos, NM, USA **2** USDA Forest Service Center for Forest Disturbance Science, Athens, GA, 30602, USA

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

Fire behavior modeling is a critical tool for understanding and predicting fire spread across landscapes, informing risk assessment, planning prescribed burns, and developing mitigation strategies. QUIC-Fire is a coupled fire-atmospheric modeling tool designed to rapidly simulate the complex interactions between fire, fuels, and atmosphere that are essential for predicting wildland fire behavior (Linn et al., 2020). QUIC-Fire simulations require preparing numerous input files with complex interdependencies and generate large volumes of output data in a variety of formats.

Here we introduce `quicfire-tools`, a Python package that provides a streamlined interface for creating, managing, and analyzing QUIC-Fire simulations. The package handles two primary aspects of the QUIC-Fire workflow: (1) programmatic creation and management of input file decks with validation and documentation, and (2) processing of simulation outputs into standard data structures compatible with the scientific Python ecosystem. By simplifying these tasks, `quicfire-tools` enables researchers, fire managers, and modelers to focus on scientific questions rather than the technical details of file format specifications.

Statement of need

Physics-based fire behavior models, such as QUIC-Fire, produce high-fidelity simulations of wildland fire behavior but present significant barriers to entry due to their complex input requirements and output formats. Users of QUIC-Fire must navigate more than 15 interdependent input files, each with dozens of parameters governing aspects like terrain, fuel characteristics, ignition patterns, and weather conditions. Manipulating these files manually is error-prone and time-consuming, particularly when setting up parameter studies or batch simulations.

Similarly, processing QUIC-Fire outputs poses challenges as the model produces binary files in various formats (compressed sparse arrays and gridded arrays) that must be properly interpreted to extract meaningful data. QUIC-Fire generates outputs for metrics such as energy release to atmosphere, reaction rate, fuel density, wind components, and fuel moisture, all of which require understanding of file formats, grid specifications, and coordinate transformations to interpret correctly.

Despite these challenges, no standardized tools existed to programmatically manage QUIC-Fire simulations before `quicfire-tools`. While ad-hoc scripts were previously used, these were neither standardized, tested, nor comprehensively documented, leading to duplicated efforts and potential inconsistencies across research groups.

`quicfire-tools` addresses these needs by providing:

1. A consistent, validated interface for creating and modifying QUIC-Fire input files, with particular attention to the complex interdependencies between parameters

2. Programmatic management of spatially and temporally varying parameters like fuels, winds, and topography
3. Robust output processing capabilities that convert binary data to standard formats (NumPy arrays, Zarr archives, netCDF files)
4. Integration with the scientific Python ecosystem (NumPy, Dask, Pandas, Xarray)
5. Comprehensive documentation with step-by-step tutorials and examples

By standardizing these essential workflow components, `quicfire-tools` enables users to focus on the scientific aspects of fire simulation rather than technical implementation details. The package facilitates more complex simulation studies, enhances reproducibility, and lowers the barrier to entry for new QUIC-Fire users.

Key Features

Input Management

The inputs module provides a comprehensive interface for creating and managing QUIC-Fire input decks, using Pydantic data models for validation and representation (Colvin et al., 2025). This approach solves two critical problems:

1. **Input file validation:** By leveraging Pydantic's validation capabilities, each parameter is checked against allowable ranges and types, preventing invalid input states before the simulation runs.
2. **Unified representation:** Instead of manually maintaining 15+ separate input files with complex interdependencies, `quicfire-tools` provides a single `SimulationInputs` object that encapsulates the entire simulation state, which can be serialized to JSON or written to individual input files to run the simulation.

A typical workflow for creating a QUIC-Fire simulation with the inputs module:

```python
from quicfire_tools.inputs import SimulationInputs

# Create a basic simulation with key parameters
simulation = SimulationInputs.create_simulation(
    nx=200,                 # Number of grid cells in x-direction
    ny=200,                 # Number of grid cells in y-direction
    fire_nz=1,              # Number of grid cells in z-direction
    wind_speed=1.7,         # Wind speed in m/s
    wind_direction=90,      # Wind direction in degrees from north
    simulation_time=600     # Simulation duration in seconds
)

# Customize fuel parameters in a uniform fuel bed
simulation.set_uniform_fuels(
    fuel_density=0.7, fuel_moisture=0.10, fuel_height=1.0
)

# Set ignition pattern
simulation.set_rectangle_ignition(
    x_min=150, y_min=100, x_length=10, y_length=100
)

# Specify output files
simulation.set_output_files(
    fuel_dens=True, emissions=True, qu_wind_inst=True
)
```

```
# Write input files to the simulation directory
simulation.write_inputs("path/to/simulation/directory")

# Or save the entire state to a single JSON file for version control or sharing
simulation.to_json("simulation_config.json")

# Later, the simulation can be recreated from the JSON file
restored_simulation = SimulationInputs.from_json("simulation_config.json")
```

Each input file is represented by a Pydantic model that inherits from the `InputFile` base class. The Pydantic model framework provides attribute validation rules and default values. This framework prevents users from creating invalid simulation states and provides error messages when invalid values are specified. The `InputFile` base class provides standardized functionality for writing input files and returning documentation for QUIC-Fire input parameters.

The package includes specialized modules for defining ignition patterns, topographic features, and wind conditions, all backed by Pydantic models for validation and consistency. The following examples illustrate how users can programmatically configure these components using `quicfire-tools`.

A suite of ignition patterns set using the `ignitions` module:

```
from quicfire_tools.ignitions import CircularRingIgnition

# Create a circular ring ignition
ignition = CircularRingIgnition(
    x_min=50, y_min=50,
    x_length=100, y_length=100,
    ring_width=20
)

# Assign to simulation
simulation.quic_fire.igntions = ignition
```

Idealized topography created using the `topography` module:

```
from quicfire_tools.topography import GaussianHillTopo

# Create a Gaussian hill topography
topo = GaussianHillTopo(
    x_hilltop=100, y_hilltop=150,
    elevation_max=50, elevation_std=15
)

# Assign to simulation
simulation.qu_topoinputs.topography = topo
```

Wind profiles defined programmatically and using imported weather station data:

```
# Add custom wind conditions programmatically
simulation.add_wind_sensor(
    wind_speeds=[5.0, 7.0, 6.0],
    wind_directions=[90, 180, 135],
    wind_times=[0, 600, 1200]
)

# Or, import from weather station data
import pandas as pd
```

```
wind_data = pd.read_csv("weather_station_data.csv")
simulation.add_wind_sensor_from_dataframe(
    df=wind_data,
    x_location=100.0,
    y_location=100.0,
    time_column="time_seconds",
    speed_column="windspeed_ms",
    direction_column="direction_deg"
)
```

Output Processing

The outputs module simplifies loading binary QUIC-Fire outputs into common Python data structures. The `SimulationOutputs` class automatically detects available output files and creates corresponding `OutputFile` instances. Each `OutputFile` class is initialized with the necessary attributes to properly load and interpret the output files, including domain dimensionality, binary file format, and metadata.

Reading and interpreting QUIC-Fire output files using the outputs module:

```
from quicfire_tools.outputs import SimulationOutputs, SimulationInputs

# Create outputs object from simulation directory
outputs = SimulationOutputs(
    "path/to/output/directory",
    fire_nz=26, ny=100, nx=100, dy=2., dx=2.
)

# Or, create the outputs object from a simulation inputs object
sim_inputs = SimulationInputs.from_json("simulation_config.json")
outputs = SimulationOutputs.from_simulation_inputs("path/to/output/directory")

# List available output variables
output_names = outputs.list_outputs()

# Get fuel density output as NumPy array
fuel_density_np = outputs.to_numpy("fuels-dens")

# Access specific timestep
timestep = 0
first_timestep_data = outputs.to_numpy("fuels-dens", timestep)
```

Output files saved in various data formats for further analysis:

```
# Get an output file object
output_file = outputs.get_output("fuels-dens")

# Save to netCDF
output_file.to_netcdf("output_directory")

# Save to zarr with chunks every 10 timesteps
output_file.to_zarr("zarr_directory", chunk_size={"time": 10})
```

Implementation

`quicfire-tools` is implemented in Python using a modular design centered around two main components:

1. The `SimulationInputs` class manages QUIC-Fire input files, each represented by specialized classes that inherit from the `InputFile` base class. These classes use Pydantic for validation, ensuring parameter values are within acceptable ranges. The use of Pydantic provides several key benefits:
   - Automatic validation of input parameters against physical constraints
   - Type checking and conversion of input values
   - Self-documenting models with clear parameter descriptions
   - Serialization/deserialization to/from JSON for storage and sharing
   - Extensibility through inheritance for specialized input types
2. The `SimulationOutputs` class manages QUIC-Fire output files, providing methods to extract data from both compressed and gridded binary formats. It includes functionality to convert outputs to standard formats like NumPy arrays, Dask arrays, netCDF files, and Zarr archives. The output processing system handles:
   - Automatic detection of available output files
   - Mapping between sparse compressed formats and dense array representations
   - Coordinate system transformations and grid alignment
   - Lazy loading for efficient memory management with large datasets

The package includes robust validation to prevent common errors, comprehensive documentation of available parameters, and extensive examples demonstrating typical workflows. Integration with scientific Python libraries ensures compatibility with common data analysis pipelines.

Conclusion

`quicfire-tools` simplifies the creation, management, and analysis of QUIC-Fire simulations, enabling users to programatically and interact with the inputs and outputs. By defining a consistent framework for representing input files, the package provides a user-friendly python class structure that facilites integration of QUIC-Fire with fuel modeling platforms (Marcozzi et al., 2025) and large ensemble applications (Ahmed et al., 2024). Furthermore, its streamlined processing of QUIC-Fire output files aids with data analysis, visualization, and communication of results. The flexible framework of `quicfire-tools` also allows for the package to keep pace with QUIC-Fire's active development.

Acknowledgements

Our thanks goes to David Robinson, Rod Linn, and the rest of the QUIC-Fire development team. We are also grateful for the input and testing from the QUIC-Fire community of practice, including Julia Oliveto, Sophie Bonner, Jay Charney, Leticia Lee, Alex Massarie, Mary Brady, and many others.

References

Ahmed, H., Shende, R., Perez, I., Crawl, D., Purawat, S., & Altintaş, İ. (2024). Towards an Integrated Performance Framework for Fire Science and Management Workflows. *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, 78–83. https://ieeexplore.ieee.org/abstract/document/10707375/

Colvin, S., Jolibois, E., Ramezani, H., Garcia Badaracco, A., Dorsey, T., Montague, D., Matveenko, S., Trylesinski, M., Runkle, S., Hewitt, D., Hall, A., & Plot, V. (2025).

Pydantic (Version v2.11.2). https://docs.pydantic.dev/latest/

Linn, R. R., Goodrick, S. L., Brambilla, S., Brown, M. J., Middleton, R. S., O'Brien, J. J., & Hiers, J. K. (2020). QUIC-fire: A fast-running simulation tool for prescribed fire planning. *Environmental Modelling and Software*, *125*, 104616. https://doi.org/10.1016/j.envsoft.2019.104616

Marcozzi, A., Wells, L., Parsons, R., Mueller, E., Linn, R., & Hiers, J. K. (2025). FastFuels: Advancing wildland fire modeling with high-resolution 3D fuel data and data assimilation. *Environmental Modelling & Software*, *183*, 106214. https://doi.org/10.1016/j.envsoft.2024.106214