1 Tstrippy

This project is built on `f2py`, which allows integration between Fortran and Python. The core motivation behind this choice is performance: Fortran, as a compiled language, provides significantly faster execution for numerically intensive tasks, while Python—especially within Jupyter notebooks—offers a convenient environment for development, experimentation, and visualization. `F2py` stands for *Fortran to Python* [Peterson, 2009], and it is included as a module within NumPy [NumPy Developers, 2025, Harris et al., 2020]. The name of the project, `tstrippy`, stands for `T`idal `Strip`ping in `P`ython.

`F2py` supports Fortran 77, 90, and 95 standards, so we chose to write the code in Fortran 90 to make use of *modules*. In Fortran, a module encapsulates data and subroutines in a manner somewhat analogous to classes in object-oriented programming. However, Fortran modules do not support inheritance, and only a single instance of a module can exist at a time—unlike classes, which can be instantiated multiple times.

`Tstrippy` package is structured around five core Fortran modules, each responsible for a distinct aspect of the simulation:

- `integrator`: This is the central module of the code. It stores particle positions and velocities, computes forces, and evolves the system forward in time. It also handles the writing of output data at specified intervals and interfaces with all other modules in the code.

- `potentials`: This module defines the analytical potentials used to compute gravitational forces. It currently supports several models, including `Plummer`, `Hernquist`, `AllenSantillian`, `MiyamotoNagai`, the bar model `LongMuraliBar` from Long and Murali [1992], and the composite model from `pouliasis2017pii` [Pouliasis et al., 2017]. This module can also be called in Python, allowing users to call potential functions directly, e.g., for computing energies during post-processing.

- `hostperturber`: This module handles the host globular cluster. It stores its orbit (i.e., timestamps, positions, and velocities) and ensures its synchronization with the simulation's internal clock. It computes the gravitational influence of the host cluster on each star particle. This is an internal module only.

- `perturbers`: Similar in function to `hostperturber`, this module supports additional perturbing clusters. It allows for the inclusion of multiple perturbers and computes their collective force on each particle. If object-oriented programming were available in Fortran, both this module and `hostperturber` would naturally inherit from a shared parent class. This module only uses the positions, masses, and characteristic radii of the other perturbers. The velocities are not imported.

- `galacticbar`: This module stores parameters for the Galactic bar, including the polynomial coefficients for its angular displacement as a function of time:

  $$\theta(t) = \theta_0 + \omega t + \dot{\omega} t^2 + \ddot{\omega} t^3 + \dots$$

  If a user wants a bar with a constant rotation speed, then they may pass two coefficients. If they want a bar that accelerates for decelerates, they may mass more coefficients for the higher order terms. The module performs transformations into the rotating bar frame, computes the forces in that frame, and then transforms the forces back to the Galactocentric reference frame.

This modular structure makes it straightforward to extend the code by adding new physics in the form of additional modules.

To make the package installable and easy to distribute, I initially followed the guide by Bovy [2025], which describes how to create a Python package using `setuptools`—the standard build system in the Python ecosystem. However, compatibility between `setuptools` and `f2py` was broken starting with NumPy $>$ 1.22 (released June 22, 2022[1]) and Python $>$ 3.9.18 (released June 24, 2024[2]). This meant that Fortran extensions could only be compiled using deprecated versions of both. These older versions of NumPy were also not compatible with Apple's ARM-based M1 and M2 processors, rendering the code unusable on modern Mac systems.

[1] `https://github.com/numpy/numpy/releases/tag/v1.22.0`
[2] `https://www.python.org/downloads/release/`

This limitation stemmed from the deprecation and eventual removal of `numpy.distutils`, the tool that previously enabled seamless integration of Fortran code in NumPy-based packages. As of NumPy 1.23 and later, `numpy.distutils` was deprecated, and with NumPy 2.0, it was removed entirely. The NumPy developers recommended migrating to `meson` [Meson Developers, 2025] or `Cmake`.

To address these issues, I migrated the build process to `meson`, a language-agnostic build system capable of compiling Fortran, C, and Python extensions across architectures. This eliminated compatibility problems and made the build process architecture-independent. The build system now automatically detects the system architecture and compiles accordingly.

The code is fully open source and available on GitHub.[3] To support users, I created documentation hosted on `readthedocs.io`[4], which includes working examples and basic usage guides. A minimal test suite, written using `pytest`, is also included. While not exhaustive, the tests ensure that core functionality remains intact as the code evolves. In the next section, I present a minimal example of how the user may use the code within a python script.

1.1 Minimum example

If the package is properly installed on the system, it can be imported at the top of any Python script:

```
import tstrippy
```

Next, the user must load or define the initial conditions. The code provides:

- The masses, sizes, and kinematics of the globular cluster catalog from Baumgardt and Hilker [2018];
- The galactic potential parameters for model II of Pouliasis et al. [2017]; and
- A galactic reference frame.

```
GCdata      = \
    tstrippy.Parsers.baumgardtMWGCs().data
MWparams    = \
    tstrippy.Parsers.potential_parameters.pouliasis2017pii()
MWrefframe  = \
    tstrippy.Parsers.potential_parameters.MWreferenceframe()
```

The user must then select the system to integrate. For example, to integrate the orbits of observed globular clusters, one must convert the ICRS coordinates to a Galactocentric frame using `astropy` and the provided MW reference frame. Alternatively, to simulate a star cluster, one can generate a Plummer sphere:

```
xp,yp,zp,vxp,vyp,vzp=\
tstrippy.ergodic.isotropicplummer(G, massHost, halfmassradius, NP)
```

Here, `NP` is the number of particles, `halfmassradius` is the system's half-mass radius, `massHost` is the total mass of the Plummer sphere, and `G` is the gravitational constant. All values must be in the same unit system.

The integrator must then be initialized. All parameters are passed via lists that are unpacked at the function call. Here is an example of initializing the integrator for a stellar stream in a potential that includes a rotating galactic bar:

```
tstrippy.integrator.setstaticgalaxy(*staticgalaxy)
tstrippy.integrator.setinitialkinematics(*initialkinematics)
tstrippy.integrator.setintegrationparameters(*integrationparameters)
tstrippy.integrator.inithostperturber(*hostperturber)
tstrippy.integrator.initgalacticbar(*galacticbar)
tstrippy.integrator.setbackwardorbit()
```

- `setstaticgalaxy` specifies the static potential model and passes its parameters.
- `setinitialkinematics` provides the initial positions and velocities of the particles.
- `setintegrationparameters` defines the initial time, timestep, and number of steps.

[3] `https://github.com/salvatore-ferrone/tstrippy`
[4] `https://tstrippy.readthedocs.io/en/latest/`

- `inithostperturber` specifies the globular cluster's trajectory and mass as a function of time.
- `initgalacticbar` defines a rotating bar. It takes the name of the bar model, potential parameters, and spin parameters.
- `setbackwardorbit` reverses the velocity vectors and sets the internal clock to count down: $t_i = t_0 - i \cdot \Delta t$. For the usecase presented in this work, `setbackwardorbit` is used for computing the globular cluster orbits and not for the star-particles.

The user can choose between two output modes during integration:

```
tstrippy.integrator.initwriteparticleorbits(nskip,myoutname,myoutdir)
tstrippy.integrator.initwritestream(nskip,myoutname,myoutdir)
```

Conceptually, these represent two output paradigms:

- `initwriteparticleorbits` saves the full orbit of each particle to an individual file.
- `initwritestream` saves full snapshots of all particles at selected timesteps.

Both functions take:

- `nskip`: number of timesteps to skip between outputs;
- `myoutname`: the base file name;
- `myoutdir`: the output directory.

The output files will be named like: `../dir/temp0.bin`, `../dir/temp1.bin`, ..., up to `../dir/tempN.bin`, where $N = N_{\text{step}}/N_{\text{skip}}$. Note that the files are written in Fortran binary format. Although `scipy.io.FortranFile` can read them, I use a custom parser based on `numpy.frombuffer` to avoid the SciPy dependency. Once all parameters are set, the user can proceed with integration using one of two methods:

Full orbit integration (in memory)

```
xt,yt,zt,vxt,vyt,vzt=\
    tstrippy.integrator.leapfrogintime(Ntimestep,nObj)
timestamps=\
    tstrippy.integrator.timestamps.copy()
```

`leapfrogintime` stores the full orbit of each particle in memory. This is useful for a small number of particles or short integrations—e.g., rapid parameter studies in a notebook. However, for large simulations it can be prohibitively memory-intensive. For instance, integrating all globular clusters at high time resolution might require:

$$7 \times N_p \times N_{\text{step}} \times 8 \text{ Byte} \approx 450 \text{ GB} \tag{1}$$

if $N_{\text{step}} \approx 10^7$. This will likely exceed system RAM.

Final state only

```
tstrippy.integrator.leapfrogtofinalpositions()
xf  = tstrippy.integrator.xf.copy()
yf  = tstrippy.integrator.yf.copy()
zf  = tstrippy.integrator.zf.copy()
vxf = tstrippy.integrator.vxf.copy()
vyf = tstrippy.integrator.vyf.copy()
vzf = tstrippy.integrator.vzf.copy()
finaltime=tstrippy.integrator.currenttime.copy()
```

`leapfrogtofinalpositions()` performs the integration but only returns the final phase-space coordinates. These arrays must be copied before deallocating memory:

```
tstrippy.integrator.deallocate()
```

Deallocating is necessary to avoid memory leaks or crashes in Jupyter when rerunning code cells.

1.2 Reflection on developing `tstrippy`

The earliest version of this code began as a simple Fortran script built to integrate test particles in specific gravitational potential models. Since I was already familiar with Fortran and had working routines, I chose to build on that foundation, gradually transforming the script into a modular and more reusable package. Rather than switching to C++ or a Python-only solution, I continued using Fortran in combination with `f2py`.

One of the primary motivations for writing my own code was flexibility. When I attempted to implement a particle spray method in *Galpy*, I found that performance degraded significantly when using custom potentials not constructed from its internal C++ backend. For example, I wanted to use the `AllenSantillian` halo model, which is not natively supported. I followed the documentation and implemented a class for it. However, custom potentials bypass `Galpy`'s optimized C++ backend, resulting in slow computations and rendering actions uncomputable (or at least with the functions I tried). This pushed me to continue developping `tstrippy`. I had a similar experience with `Amuse`.

This choice, however, came with challenges. At one point, I tried implementing potentials derived from exponential density profiles, which do not admit closed-form solutions for the potential. I attempted to work in elliptical coordinates, motivated by the idea that the potential would depend only on the "distance" from the center along the equipotential surfaces. While I successfully implemented simple potential models in elliptical coordinates, I naïvely overlooked that this changes the equations of motion entirely due to the underlying geometry. I found that my orbits always diverged (except in special cases). It was only later that I realized I would need to account for the metric tensor and Christoffel symbols to properly integrate orbits in elliptical coordinates. At that point, I chose not to pursue this further, prioritizing scientific analysis and launching other simulations instead. This experience helped me appreciate why codes like `Agama` and many published works prefer basis function expansions for such problems. In hindsight, this episode was a perfect example of how even unsuccessful attempts can lead to valuable insights. Below, I summarize some of the key advantages and limitations I encountered while developing a code from scratch to answer a scientific question.

1.2.1 Advantages and Limitations

Developing and maintaining this codebase brought several clear benefits:

- *Understanding.* Writing the code forced me to deeply understand the modeling techniques involved. Otherwise, the results would have been incorrect.
- *Flexibility.* I could implement exactly the models I needed and extend them as required.
- *Transparency.* Results produced by my code can be verified and reproduced by others.
- *Reusability.* Ongoing development helped me uncover and fix subtle bugs (e.g., related to non-symplectic integrators) that do not trigger obvious errors and only appear in edge cases or after close inspection. Long-term engagement with the code is key to catching these issues.
- *Collaboration.* A fellow researcher at the Paris Observatory is now using the code. I also used it to supervise a master's student during their semester research project, something that would not have been possible without building a user-friendly tool.
- *Growth.* This project pushed me to adopt best practices: version control, documentation, modularity. Developing my own code has also made it easier to understand external libraries. For example, when I implemented the King model, I studied `Galpy`'s internals to cross-check my own method. I was delighted by how much easier it was for me to use, despite the fact that I had not touched it within a year.

However, there were also drawbacks:

- *Time cost.* Developing the code took time away from direct scientific analysis. It's possible I could have performed more simulations or performed more analyses had I not develop my code to such an extent.
- *Feature limitations.* My code still lacks capabilities present in other packages: such as basis function expansions, action-angle variable computation, or parallelization strategies using MPI or OpenMP.

- *Changing relevance.* Scientific priorities and available tools evolve rapidly. A general-purpose tool may become obsolete more quickly than a single-use script written for a specific question.
- *Compatibility and maintenance burden.* Making the code accessible to other users also introduces challenges related to cross-platform compatibility and dependency management. Software environments evolve, compilers are updated, dependencies can deprecate, and build tools change. Even with the help of modern tools like `meson` or `f2py`, ensuring continued compatibility requires regular testing and adaptation. As the codebase grows, the maintenance load increases, and sustaining it as a single developer becomes increasingly difficult, especially if the tool is made too general.

Nonetheless, the code was designed to address concrete scientific questions about Milky Way stellar streams and globular clusters. In the next two chapters, I present how this tool contributed to advancing our understanding of the Galaxy.

References

H. Baumgardt and M. Hilker. A catalogue of masses, structural parameters, and velocity dispersion profiles of 112 Milky Way globular clusters. MNRAS, 478(2):1520–1557, August 2018. doi: 10.1093/mnras/sty1057.

Jo Bovy. Python packaging user guide. `https://pythonpackaging.info/`, 2025. Accessed: 2025-08-06.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. Nature, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2.

Kevin Long and Chigurupati Murali. Analytical Potentials for Barred Galaxies. ApJ, 397:44, September 1992. doi: 10.1086/171764.

Meson Developers. The meson build system: Manual and reference documentation. `https://mesonbuild.com/index.html`, 2025. Accessed: 2025-08-06.

NumPy Developers. F2py user guide and reference manual. `https://numpy.org/doc/stable/f2py/`, 2025. Accessed: 2025-08-06.

Pearu Peterson. F2py: a tool for connecting fortran and python programs. *International Journal of Computational Science and Engineering*, 4(4):296–305, 2009. doi: 10.1504/IJCSE.2009.029165.

E. Pouliasis, P. Di Matteo, and M. Haywood. A Milky Way with a massive, centrally concentrated thick disc: new Galactic mass models for orbit computations. A&A, 598:A66, February 2017. doi: 10.1051/0004-6361/201527346.