Introduction

This document accompanies a code refactor into modules, intended to make the underlying code easily generalisable to a broader class of potentials.

1 Refactor Summary

1.1 Potentials.py

The file defines a framework for scalar-field inflationary potentials.

An abstract base class, `Potential`, represents a generic potential $V(\phi)$. Subclasses are required to implement a method $V(\phi) = \texttt{value}(\phi)$, which is aliased as the call method here.

To ensure that generic potentials can be used, a numerical approximation is implemented at this level of abstraction as well. The derivative is estimated using a central finite difference,

$$V'(\phi) \approx \frac{V(\phi + h) - V(\phi - h)}{2h},$$

where the step size h is recursively halved until the difference between successive approximations is smaller than 10^{-5}. The final converged value is returned.

A concrete subclass, `PowerPotential`, implements a monomial potential of the form $V(\phi) = V_0\phi^m$,, where m is the power and V_0 is a normalization constant. For this potential, the derivative is implemented analytically as $V'(\phi) = mV_0\phi^{m-1}$.

1.2 Integrate.py

This file provides numerical integration utilities for inflationary dynamics in the Hamilton–Jacobi (HJ) formalism. Its primary role is to integrate the Hubble parameter $H(\phi)$ as a function of the scalar field ϕ, while tracking the slow-roll parameter ϵ and, optionally, the number of e-folds N.

Thinking about this code as something to try-out for future numerical integration problems, a small hierarchy of derivative classes is defined, it's main application being to have two objects for each the positive and negative root of the $\frac{3}{2}H^2 - \frac{1}{2}V(\phi)$.

The abstract base class `DerivativeBase` defines a callable interface for evaluating a derivative given a pair of variables. The class `DerivativeFromFunction` wraps a generic function of two variables into this interface.

Two concrete Hamilton–Jacobi derivatives are implemented. The class `HJDerivative` evaluates the positive branch of the Hamilton–Jacobi equation, while `NegativeHJDerivative` implements the corresponding negative branch. The magnitude of the gradient is computed through helper functions which evaluate $(H')^2 = \frac{3}{2}H^2 - \frac{1}{2}V(\phi)$, and return its square root with appropriate sign handling.

The slow-roll parameter ϵ is evaluated in two forms. The exact Hamilton–Jacobi expression, $\varepsilon = \frac{2\,H'^2}{H^2}$, is used during numerical integration, and is used alongside the slow-roll approximation, $\varepsilon_{\mathrm{SR}} = \frac{1}{2}\left(\frac{V'(\phi)}{V(\phi)}\right)^2$, to detect entry into the slow-roll regime, with helper functions for each.

In this refactoring, the RK4 step is implemented as a function of ϕ, H, the `Derivative` (positive or negative root for H') and the step size (which is adapted as a function of ε, as before).

The function `get_trajectory_decreasing_phi` matches the old `trajectory` function, except refactored using the abstractions written for RK4, potentials and `Derivative`, and the option to track N. The function `get_trajectory_increasing_phi` considers the scheme for the time-reversing direction. For simplicity, this function uses limits on ε and $\varepsilon_{\mathrm{SR}}$ to determine when to evoke the Slow Roll quadrant crossing step, rather than waiting for an RK4 error to be raised, which the legacy `tot_traj`.