Introduction

Python is a particularly appealing language to carry out data analysis, owing in part to its user-friendly character as well as its access to well maintained and powerful libraries like NumPy and SciPy. Still, for the purpose of analyzing data in a lattice QCD context, some desirable functionality is missing from these libraries. Moreover, scripting languages tend to be slower than compiled ones. To help address these points we present the LatticeToolbox [1], a collection of Python modules to facilitate lattice QCD data analysis. Modules are sped up behind the scenes using Numba and parallelizers.

Motivation and strategy

The LatticeToolbox was originally developed by H. Sandmeyer in the context of HotQCD projects. Taking a cue from other open data movements like the ILDG, we have refactored the code, improved its performance, then made it publicly available on GitHub [1]. As part of the refactoring, we try to modularize as shown below. We have a large set of unit tests which we regularly run to ensure the code is robust against changes. As a modest step towards interoperability, we dedicate a section of the code to interfacing with configuration binaries and other lattice software.

Interfacing

Having evolved in the context of HotQCD and MILC projects, the code interfaces with some software and conventions of these groups. We also try to make the code flexible to conventions in the broader lattice community. For instance:

- Reading in gauge configurations (NERSC, eventually ILDG)
- Jackknifing of C. Schmidt’s DenseCode output
- Reading .gpl files from P. LePage’s tools

Examples of streamlined coding

Listing 1: An example of how the LatticeToolbox can be used to carry out a simple hadron resonance gas computation of \( v_0^{\beta} \). As one can see from the gen\_chi call, arbitrary conserved-charge cumulants are supported.

```
import numpy as np
import latqcdtools.physics as physics
from latqcdtools.math import get_err_str
from latqcdtools.physics.HRG import HRG

# Write terminal output to log file. Includes git commit hash.
logger.info('Computing chi2B.
# This computation is vectorized since T is a numpy array.

r0 = latqcdtools.physics.lattice_params
# Read in Polyakov loop measurements, data = readTable("data.txt"); data

beta: = 0.12

# Create array of temperatures in physical units
T = np.linspace(T[0],T[-1],1001)

# Error in Tc estimate comes from 1000 Gaussian bootstrap samples
Tcr0_lit = 0.7457
Tcr0e = r0.muT(year=2014, units="MeVinv") * result_err[0]
Tcr0 = r0.muT(year=2014, units="MeVinv") * result[0]

# Perform CI(\(\beta\)) continuum-limit extrapolation
result = continuumExtrapolate(Nt, Tc, show_results=True, plot_results=True, xtype="Nt",
plot_results=True, ytext="Tc [MeV]", ytitle="CI extrapolation")
```

Listing 2: Given are results for \( \langle P_i \rangle \) at various \( N_t \) from pure SU(3) lattice calculations. This code (1) estimates the inflection point of \( \langle P_i \rangle \) as a function of \( T \) to get \( T_c \); (2) performs a parallelized bootstrap of (1) to get \( \sigma_T \); (3) repeats for all \( N_t \) and performs a continuum-limit extrapolation; and (4) compares \( T_c \) against the literature result using a Z-test. The bootstrapping method is analogistic to the to-be-bootstrapped function.

Physics modules

We conclude with some physics modules that might be of interest both to lattice practitioners and those studying QCD phenomenology:

- HotQCD parametersizations of, e.g. \( a f_K(\beta), r_2 m_s(\beta) \)
- Physical parameters and their errors, e.g. \( m_v, m_p \)
- Hadron Resonance Gas model [2]
- QCD Equation of State [3]
- Static quark potential and Polyakov loop observables
- We will continue adding more!

References