MPS on Probability

This program studies the stochastic process problems with an algorithm called the matrix product state (MPS) that showed its power first in many body quantum physics. Noticing the analogy between probability theory and quantum mechanics, we adapt the algorithm to deal with many variable stochastic process problems.

We can use transitional matrix methods to study stochastic processes and extract a lot of interesting properties. However, just like in the many body quantum system, the transitional matrix grows exponentially in a system of multiple degrees of freedom and becomes intractable with 30~50 variables. Nervertheless, many models benefit from the property of locality and it is proved that, writing states as product of small matrices can effectively reduce the system complexity. We write transitional matrices as well as state vectors in terms of product of matrices and solve the time evolution problem efficiently.

A detailed description of the problems, models and the algorithm is in the documentation file Project_note.pdf and here is a quick start guide.

Structure:

- **proba.py** is an example main caller and you can tune the parameters in the main function which runs both exact and mps models. It calculates the joint probability, the mean value and the variance in different models at any time point. And it can compare the results of mps solution and exact solution for short chains. Please be aware that the exact method can only calculate a short chain of a typical maximum size of 14, and the program raises an exception if the chain is too long to be solved exactly.
- **proba_mps.py** is an example that only uses the MPS method. Its function is similar to proba.py but does not calculate an exact solution. You can feel free to try a system of size 200. It has four models and the user can choose which model to use.
- **model.py** is the base model class for all the model classes.
- **solver.py** is the base solver class.
- **exactsolver.py** defines the exact traditional matrix method.
- **mpssolver.py** defines the MPSsolver for the matrix product states method. It has the several parameters that can be tuned. However we do not advise users to change these parameters unless exceptions of local minimum are encountered.
- **measurement.py** defines the base measurement class Measurement for all the measurement classes. It can compute the joint probability, correlation function, variance, and mean value at any time point. Users need to define a measurement task first.
- **mpsmeasurement.py**: It is the derived measurement class for MPS. It has the following assumptions: for a mps, physical index 0 represents down and physical index 1 represents up. solver.results has elements time ordered and start from time 0. the sites start from 0. The first mps is of shape (2,1,n), and the last mps of shape (2,n,1).
- **exactmeasurement.py**: It is the derived measurement class for exact solution.

Usage

Proba.py is a very nice example of the usage.

- Step 1: Define a model. Each model has its own parameters and all examples are shown in the file. For example:

```
angry_boys = AngryBoys(size = 10, remain_proba = 0.1, init_state = "all down")
```

defines an AngryBoy model with length 10, 10% probability to remain unchanged and the initial state is that everyone is down (not angry).

- Step 2: Define solvers. For example:

```
exact_solver = ExactSolver(model = angry_boys)
```

defines an exact solver that contains a reference to the model.

```
mps_solver = MpsSolver(model = angry_boys, bound_dimension = 10)
```

defines an MPS solver of the same model and bound dimension 10.

- Step 3: Define measurements. For example:

```
exact_measurement = ExactMeasurement(exact_solver)
```

and

```
mps_measurement   = MpsMeasurement(mps_solver)
```

define the corresponding measurement objects.

- Step 4: Add measurement tasks

```
mps_measurement.addMeasureTask(task)
```

adds a specific measurement task. The task object is formed as a tuple that contains the measurement type, time point (not specifying time point corresponds to the last time) and the sites the user is interested in. Some examples are listed below:

```
("Correlation", 1, [1, 2, 3, 4, 5])
```

means the average of S1*S2*S3*S4*S5 at t = 1

```
("Correlation",  [1, 2, 3, 4, 5])
```

means the avearge of S1*S2*S3*S4*S5 at the last time point

```
("Mean", 1, [-1])
```

means the average of site N at t = 1, for a chain of length N

```
("Variance", 1, [1])
```

means the variance of site 1 at t = 1.

```
("Proba", [(1, "up"), (3, "down")])
```

means the probability P(S1 == up AND S2 == down).

- Step 5: Call the measure function. For example

```
mps_measurement.measure()
```

and then the measured results are contained in

```
mps_measurement.measure_result_list
```

. Feel free to use your favorite plotting function to make pretty plots of the

```
measure_result_list
```

! Remember to call

```
mps_measurement.clearMeasurement()
```

before you make new measurements.

Exceptions:

User should be aware that when calculating certain models using MPS solver, he will possibly encounter an exception of negative norm. The reason for the complication is rooted in the compression algorithm. Indeed when compressing an MPS, one can not guarantee that the compressed MPS can be normalized to a probability distribution. The L-1 norm of the compressed MPS might be negative. Such problem, which is an inherited limit of the algorithm itself, cannot be tackled at the level of implementation.

Authors:

- Bin Xu (binarybin)
- Peiqi Wang (pqwang1026)
- Liangsheng Zhang (phzls)
- Peter Gjeltema (PJ)
- Jun Xiong (xiongPU)

Licence