The Journal of Open Source Software

QuantEcon.py: A community based Python library for quantitative economics

Quentin Batista[5], Chase Coleman[4], Yuya Furusawa[6], Shu Hu[1], Smit Lunagariya[3], Spencer Lyon[4], Matthew McKay[1], Daisuke Oyama[2], Thomas J. Sargent[4], Zejin Shi[7], John Stachurski[1], Pablo Winant[8], Natasha Watkins[1], Ziyue Yang[1], and Hengcheng Zhang[1]

1 The Australian National University **2** University of Tokyo **3** Indian Institute of Technology (BHU), Varanasi **4** New York University **5** Massachusetts Institute of Technology **6** Crop.inc **7** The University of Arizona **8** ESCP Business School and Ecole Polytechnique

DOI: 10.21105/joss.05585

Software

- Review
- Repository
- Archive

Editor: Sebastian Benthall

Reviewers:

- @janosg
- @mnwhite

Submitted: 20 May 2023
Published: 06 January 2024

Summary

Economics traditionally relied on tractable mathematical models, diagrams, and simple regression methods to analyze and understand economic phenomena. However, in recent decades, economists have increasingly shifted towards more computationally challenging problems, involving large numbers of heterogeneous agents and complex nonlinear interactions.

QuantEcon.py is an open-source Python library that helps to support this shift towards more computational intensive research in the field of economics. First released in 2014, QuantEcon.py has been under continuous development for around 9 years. The library includes a wide range of functions for economic analysis, including numerical methods, data visualization, estimation, and dynamic programming, implementing a number of fundamental algorithms used in high performance computational economics. In this article we review the key features of the library.

Statement of Need

Economists use a variety of economic, statistical and mathematical models as building blocks for constructing larger and more fully-featured models. Some of these are relatively unique to economics and finance. For example, many macroeconomic and financial models include a stochastic volatility component, since asset markets often exhibit bursts of volatility. Other building blocks involve optimization routines, such as firms that maximize present value given estimated time paths for profits and interest rates. Firms modeled in this way are then plugged into larger models that contain households, banks and other economic agents.

QuantEcon.py focuses on supplying building blocks for constructing economic models that are fast, efficient and simple to modify. This encourages code re-use across the economics community, without enforcing particular model structure through a top-down development process.

Implementation Choices

In terms of software systems and architecture, QuantEcon.py is built on top of standard libraries such as NumPy (Harris et al., 2020) and SciPy (Virtanen et al., 2020), while also heavily leveraging Numba (Lam et al., 2015) for just-in-time (JIT) code acceleration, combined with automatic parallelization and caching when possible. (Numba is a just-in-time (JIT) compiler for Python first developed by Continuum Analytics that can generate optimized LLVM machine

code at run-time.) JIT-based acceleration is essential to QuantEcon's strategy of providing code for computational economics that is performant, portable and easy to modify.

For installation and maintenance ease, QuantEcon maintainers restrict contributions to depend on libraries available in Anaconda.

The documentation is available on readthedocs.

Status

QuantEcon.py is released under the open-source MIT License and is partly maintained and supported by QuantEcon, a NumFOCUS fiscally sponsored project dedicated to development and documentation of modern open source computational tools for economics, econometrics, and decision making.

QuantEcon.py is available through the Python Package Index:

```
pip install quantecon
```

or through `conda`:

```
conda install -c conda-forge quantecon
```

Capabilities

This section gives a basic introduction of `quantecon` and its usage. The `quantecon` python library consists of the following top level modules:

- Game Theory (`game_theory`)
- Markov Chains (`markov`)
- Optimization algorithms (`optimize`)
- Random generation utilities (`random`)

The library also has some other submodules containing utility functions and miscellaneous tools such as implementations of Kalman filters, tools for directed graphs, algorithm for solving linear quadratic control, etc.

Game Theory

The `game_theory` submodule provides efficient implementation of state-of-the-art algorithms for computing Nash equilibria of normal form games.

The following snippet computes all mixed Nash equilibria of a 2-player (non-degenerate) normal form game by support enumeration and vertex enumeration:

```
>>> import quantecon as qe
>>> import numpy as np
>>> import pprint

>>> bimatrix = [[(3, 3), (3, 2)],
...             [(2, 2), (5, 6)],
...             [(0, 3), (6, 1)]]
>>> g = qe.game_theory.NormalFormGame(bimatrix)
>>> print(g)
2-player NormalFormGame with payoff profile array:
[[[3, 3],  [3, 2]],
 [[2, 2],  [5, 6]],
 [[0, 3],  [6, 1]]]
>>> NEs = qe.game_theory.support_enumeration(g)
```

```
>>> pprint.pprint(NEs)
[(array([1., 0., 0.]), array([1., 0.])),
 (array([0.8, 0.2, 0. ]), array([0.66666667, 0.33333333])),
 (array([0.        , 0.33333333, 0.66666667]), array([0.33333333, 0.66666667]))]
>>> NEs = qe.game_theory.vertex_enumeration(g)
>>> pprint.pprint(NEs)
[(array([1., 0., 0.]), array([1., 0.])),
 (array([0.        , 0.33333333, 0.66666667]), array([0.33333333, 0.66666667])),
 (array([0.8, 0.2, 0. ]), array([0.66666667, 0.33333333]))]
```

The Lemke-Howson algorithm (Codenotti et al., 2008; Lemke & Howson, 1964) is also implemented, which computes one Nash equilibrium of a 2-player normal form game:

```
>>> qe.game_theory.lemke_howson(g)
(array([1., 0., 0.]), array([1., 0.]))
>>> qe.game_theory.lemke_howson(g, init_pivot=1)
(array([0.        , 0.33333333, 0.66666667]), array([0.33333333, 0.66666667]))
```

This routine `lemke_howson` scales up to games with several hundreds actions.

For N-player games, the McLennan-Tourky algorithm (McLennan & Tourky, 2005) computes one (approximate) Nash equilibrium:

```
>>> payoff_profiles = [(3, 0, 2),
...                    (1, 0, 0),
...                    (0, 2, 0),
...                    (0, 1, 0),
...                    (0, 1, 0),
...                    (0, 3, 0),
...                    (1, 0, 0),
...                    (2, 0, 3)]
>>> g = qe.game_theory.NormalFormGame(np.reshape(payoff_profiles, (2, 2, 2, 3)))
>>> print(g)
3-player NormalFormGame with payoff profile array:
[[[[3, 0, 2],   [1, 0, 0]],
  [[0, 2, 0],   [0, 1, 0]]],

 [[[0, 1, 0],   [0, 3, 0]],
  [[1, 0, 0],   [2, 0, 3]]]]
>>> mct = qe.game_theory.mclennan_tourky(g)
>>> pprint.pprint(mct)
(array([0.61866018, 0.38133982]),
 array([0.4797706, 0.5202294]),
 array([0.37987835, 0.62012165]))
```

The `game_theory` submodule also contains implementation of several learning/evolutionary dynamics algorithms, such as fictitious play (and its stochastic version), best response dynamics (and its stochastic version), local interaction dynamics, and logit response dynamics.

Markov Chains

The `markov` module deals with computation related to Markov chains.

This module contains a class `MarkovChain` which represents finite-state discrete-time Markov chains.

```
>>> P = [[0, 1, 0, 0, 0],
...      [0, 0, 1, 0, 0],
...      [0, 0, 0, 1, 0],
```

```
...       [2/3, 0, 0, 0, 1/3],
...       [0, 0, 0, 1, 0]]
>>> mc = qe.markov.MarkovChain(P)
```

The `MarkovChain` object provides access to useful information such as:

- Whether it is irreducible:

  ```
  >>> mc.is_irreducible
  True
  ```

- Its stationary distribution(s):

  ```
  >>> mc.stationary_distributions
  array([[0.2, 0.2, 0.2, 0.3, 0.1]])
  ```

- Whether it is (a)periodic:

  ```
  >>> mc.is_aperiodic
  False
  ```

- Its period and cyclic classes:

  ```
  >>> mc.period
  2
  >>> mc.cyclic_classes
  [array([0, 2, 4]), array([1, 3])]
  ```

- Simulation of time series of station transitions:

  ```
  >>> mc.simulate(10)
  array([0, 1, 2, 3, 0, 1, 2, 3, 4, 3])
  ```

The `MarkovChain` object is also capable of determining communication classes and recurrent classes (relavant for reducible Markov chains).

It is also possible to construct a `MarkovChain` object as an approximation of a linear Gaussian AR(1) process,

$$y_t = \mu + \rho y_{t-1} + \epsilon_t,$$

by Tauchen's method (`tauchen`) (Tauchen, 1986) or Rouwenhorst's method (`rouwenhorst`) (Rouwenhorst, 1995):

```
>>> tauchen_mc = qe.markov.tauchen(n=4, rho=0.5, sigma=0.5, mu=0., n_std=3)
>>> tauchen_mc.state_values
array([-1.73205081, -0.57735027,  0.57735027,  1.73205081])
```

```
>>> rhorst_mc = qe.markov.rouwenhorst(n=4, rho=0.5, sigma=0.5, mu=0.)
>>> rhorst_mc.state_values
array([-1.        , -0.33333333,  0.33333333,  1.        ])
```

The `markov` module can also be used for representing and solving discrete dynamic programs (also known as Markov decision processes) with finite states and actions:

```
>>> R = [[5, 10],
...      [-1, -float('inf')]]  # Rewards
>>> Q = [[(0.5, 0.5), (0, 1)],
...      [(0, 1), (0.5, 0.5)]]  # Transition probabilities
>>> beta = 0.95  # Discount factor
>>> ddp = qe.markov.DiscreteDP(R, Q, beta)
```

The `DiscreteDP` class currently implements the following solution algorithms:

- value iteration;
- policy iteration;
- modified policy iteration;
- linear programming.

To solve the model:

- By the *value iteration* method:

```
>>> res = ddp.solve(method='value_iteration', v_init=[0, 0], epsilon=0.01)
>>> res.sigma  # (Approximate) optimal policy function
array([0, 0])
>>> res.v  # (Approximate) optimal value function
array([ -8.5665053 , -19.99507673])
```

- By the *policy iteration* method:

```
>>> res = ddp.solve(method='policy_iteration', v_init=[0, 0])
>>> res.sigma  # Optimal policy function
array([0, 0])
>>> res.v  # Optimal value function
array([ -8.57142857, -20.        ])
```

Similary, we can also solve the model using *modified policy iteration* and *linear programming* by changing the `method` option in `ddp.solve`.

Optimization

The `optimize` module provides various routines for solving optimization problems and root finding.

Although some methods such as `bisect` and `brentq` have been implemented in popular libraries such as SciPy, the major benefit of the `quantecon` implementation relative to other implementations is JIT-acceleration and hence they can be embedded in user-defined functions that target the Numba JIT compiler.

Linear Programming

This module contains a linear programming solver based on the simplex method, `linprog_simplex`, which solves a linear program of the following form:

$$\begin{aligned} \max_x \ & c^T x \\ \text{subject to } \ & A_{\text{ub}} x \leq b_{\text{ub}}, \\ & A_{\text{eq}} x = b_{\text{eq}}, \\ & x \geq 0. \end{aligned}$$

The following is a simple example solved by `linprog_simplex`:

```
>>> c = [4, 3]
>>> A_ub = [[1, 1],
...         [1, 2],
...         [2, 1]]
>>> b_ub = [10, 16, 16]
>>> c, A_ub, b_ub = map(np.asarray, [c, A_ub, b_ub])
>>> res = qe.optimize.linprog_simplex(c, A_ub=A_ub, b_ub=b_ub)
>>> res.x, res.fun, res.success
(array([6., 4.]), 36.0, True)
```

```
>>> res.lambd  # Dual solution
array([2., 0., 1.])
```

Scalar Maximization

The optimize module implements the Nelder-Mead algorithm (Gao & Han, 2012; Lagarias et al., 1998; Singer & Singer, 2004) for maximizing a scalar-valued function with one or more variables.

```
>>> from numba import njit
>>> @njit
... def rosenbrock(x):
...     return -(100 * (x[1] - x[0] ** 2) ** 2 + (1 - x[0])**2)
...
>>> x0 = np.array([-2, 1])
>>> res = qe.optimize.nelder_mead(rosenbrock, x0)
>>> res.x, res.fun, res.success
(array([0.99999814, 0.99999756]), -1.6936258239463265e-10, True)
```

There is also the scalar maximization function brentq_max which maximizes a function within a given bounded interval and returns a maximizer, the maximum value attained, and some additional information related to convergence and the number of iterations.

```
>>> @njit
... def f(x):
...     return -(x + 2.0)**2 + 1.0
...
>>> qe.optimize.brent_max(f, -3, 2)  # x, max_value_of_f, extra_info
(-2.0, 1.0, (0, 6))
```

Root Finding

This module also includes routines that find a root of a given function. Presently, quantecon has the following implementations:

- bisect
- brentq
- newton
- newton_halley
- newton_secant

The following snippet uses brentq to find the root of the function f in the interval $(-1, 2)$.

```
>>> @njit
... def f(x):
...     return np.sin(4 * (x - 1/4)) + x + x**20 - 1
...
>>> qe.optimize.brentq(f, -1, 2)
results(root=0.40829350427935973, function_calls=12, iterations=11, converged=True)
```

Miscellaneous Tools

The library also contains some other tools that help in solving problems such as linear quadratic optimal control and discrete Lyapunov equations, analyzing dynamic linear economies, etc. A brief overview of some of these routines is given below:

Matrix Equations

The function `solve_discrete_lyapunov` computes the solution of the discrete Lyapunov equation given by:

$$AXA' - X + B = 0.$$

```
>>> A = np.full((2, 2), .5)
>>> B = np.array([[.5, -.5], [-.5, .5]])
>>> qe.solve_discrete_lyapunov(A, B)
array([[ 0.5, -0.5],
       [-0.5,  0.5]])
```

Similarly, the function `solve_discrete_riccati` computes the solution of the discrete-time algebraic Riccati equation (Chiang et al., 2010):

$$X = A'XA - (N + B'XA)'(B'XB + R)^{-1}(N + B'XA) + Q.$$

LQ Control

The library has a class `LQ` for analyzing linear quadratic optimal control problems of either the infinite horizon form or the finite horizon form:

```
>>> Q = np.array([[0., 0.], [0., 1]])
>>> R = np.array([[1., 0.], [0., 0]])
>>> RF = np.diag(np.full(2, 100))
>>> A = np.full((2, 2), .95)
>>> B = np.full((2, 2), -1.)
>>> beta = .95
>>> T = 1
>>> lq_mat = qe.LQ(Q, R, A, B, beta=beta, T=T, Rf=RF)
>>> lq_mat
Linear Quadratic control system
  - beta (discount parameter)       : 0.95
  - T (time horizon)                : 1
  - n (number of state variables)   : 2
  - k (number of control variables) : 2
  - j (number of shocks)            : 1
```

Graph Tools

The library contains a class `DiGraph` to represent directed graphs and provide information about the graph structure such as strong connectivity, periodicity, etc.

```
>>> adj_matrix = [[0, 1, 0, 0, 0],
...               [0, 0, 1, 0, 0],
...               [0, 0, 0, 1, 0],
...               [1, 0, 0, 0, 1],
...               [0, 0, 0, 1, 0]]
>>> node_labels = ['a', 'b', 'c', 'd', 'e']
>>> g = qe.DiGraph(adj_matrix, node_labels=node_labels)
>>> g
Directed Graph:
  - n(number of nodes): 5
```

- Check if the graph is strongly connected:

```
>>> g.is_strongly_connected
True
```

- Inspect the periodicity of the graph:

```
>>> g.is_aperiodic
False
>>> g.period
2
>>> g.cyclic_components
[array(['a', 'c', 'e'], dtype='<U1'), array(['b', 'd'], dtype='<U1')]
```

Future Work

QuantEcon developers are considering future projects such as adding more equilibrium computation algorithms for N-player games and supporting extensive form games. In addition, QuantEcon aims to extend its current implementation to other backend libraries like JAX or other GPU providing libraries to utilize modern computing systems and provide faster execution speeds.

References

Chiang, C.-Y., Fan, H.-Y., & Lin, W.-W. (2010). Structured Doubling Algorithm for Discrete-Time Algebraic Riccati Equations with Singular Control Weighting Matrices. *Taiwanese Journal of Mathematics*, *14*(3A), 933–954. https://doi.org/10.11650/twjm/1500405875

Codenotti, B., De Rossi, S., & Pagan, M. (2008). An Experimental Analysis of Lemke-Howson Algorithm. *arXiv Preprint arXiv:0811.3247*.

Gao, F., & Han, L. (2012). Implementing the Nelder-Mead Simplex Algorithm with Adaptive Parameters. *Computational Optimization and Applications*, *51*(1), 259–277. https://doi.org/10.1007/s10589-010-9329-3

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Fernández del Río, J., Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array Programming with NumPy. *Nature*, *585*, 357–362. https://doi.org/10.1038/s41586-020-2649-2

Lagarias, J. C., Reeds, J. A., Wright, M. H., & Wright, P. E. (1998). Convergence Properties of the Nelder–Mead Simplex Method in Low Dimensions. *SIAM Journal on Optimization*, *9*(1), 112–147.

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A LLVM-based Python JIT Compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 1–6.

Lemke, C. E., & Howson, J. T., Jr. (1964). Equilibrium Points of Bimatrix Games. *Journal of the Society for Industrial and Applied Mathematics*, *12*(2), 413–423.

McLennan, A., & Tourky, R. (2005). *From Imitation Games to Kakutani*. University of Pennsylvania.

Rouwenhorst, K. G. (1995). Asset Pricing Implications of Equilibrium Business Cycle Models. In *Frontiers of Business Cycle Research* (pp. 294–330). Princeton University Press. https://doi.org/10.2307/j.ctv14163jx.16

Singer, S., & Singer, S. (2004). Efficient Implementation of the Nelder–Mead Search Algorithm. *Applied Numerical Analysis & Computational Mathematics*, *1*(2), 524–534. https://doi.org/10.1002/anac.200410015

Tauchen, G. (1986). Finite State Markov-Chain Approximations to Univariate and Vector Autoregressions. *Economics Letters*, *20*(2), 177–181. https://doi.org/10.1016/0165-1765(86)90168-0

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., & others. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*(3), 261–272.