econsim: Software for a Big Data Approach to Optimal Policy Problems with Heterogeneity

Christian Baker Jeremy Bejarano Richard W. Evans
Kenneth L. Judd Kerk L. Phillips

July 2013

econsim Overview

- **Reusable Python Package:** `econsim` is a Python package used to solve optimal policy problems given a heterogeneous population.
- **Non-Continuous, Non-Convex:** `econsim` is designed specifically to handle constrained optimization problems with discontinuities and non-convexity.

econsim Overview

- **Convenient User Interface:** It provides a convenient user interface for defining the household problem and allows for an easy way to define and alter the distributions involved in the social planner's problem.
- **Built To Scale:** `econsim` uses MPI and technology for managing large amounts of data and is suited to scale to large supercomputer clusters.

High Performance Computing Features

`econsim` is a parallel library that uses big data technology. Its challenges deal with the computation of many separate optimization problems and the management of a large amount of data.

- `econsim` is parallelized using MPI (Message Passing Interface)—the de facto standard for distributed memory programming.
- `econsim` is designed to scale. We currently are using the 10,000 core supercomputer at BYU's Fulton Supercomputing Lab.

High Performance Computing Features

The key feature of the `econsim` method is the reuse of the individual responses. These are saved in a large database file. Then, a new distribution over type space may be substituted and the efficient policies can be computed relatively quickly.

- This database file can get large (100s of GBs). To manage data of this size, we use HDF5.
- From the HDF Group's website, "HDF5 is a unique technology suite that makes possible the management of extremely large and complex data collections."

Points of talk

- What kinds of problems?
- How do you use it?
- How does it work?
- How well does it perform?

General family of models

- Type space: heterogeneous individuals of type $\theta \in \Theta$
  - with distribution over types $\Gamma(\theta)$
- Policy space: policies over type or other $\tau \in \mathcal{T}$
- Individual optimization: $c(\theta, \tau)$
- Policy objective:

$$\max_{\tau} \; U\Big(\mathbf{c}^*(\theta, \tau), \Gamma(\theta)\Big)$$
$$\text{s.t. } R\Big(\mathbf{c}^*(\theta, \tau), \Gamma(\theta)\Big) \geq \bar{R}$$

Big data approach

- Nice for large dimensional Θ
- Essential when τ creates nonconvex optimization

Examples of Usages

Sales Tax
Type-space: Income, Elasticities
Policy-space: Tax rate on each good
Consumption decision

Income Tax with Brackets
Type-space: Income, Elasticities
Policy-space: Income tax rate in each bracket
Labor decision

Insurance
Type-space: Health, Income
Policy-space: Premium, Deductible, Copay
Decision on Carefullness (# Insured, given costs)

Politics
Type-space: Political Leanings
Policy-space: Platform (for or against on a variety of issues)
Policicitician chooses platform

Our sales tax model: policy maker

- Policy maker chooses τ to maximize total welfare subject to revenue constraint

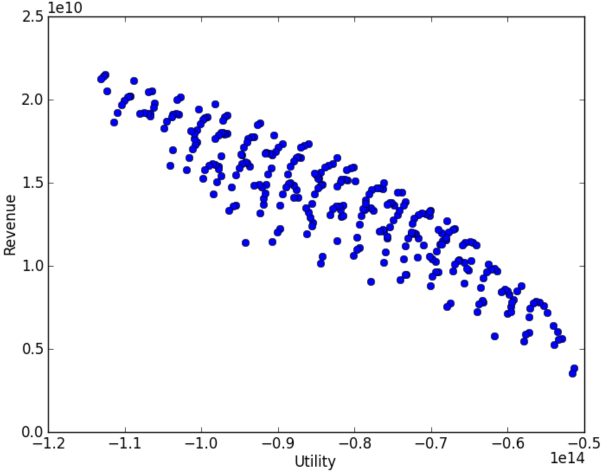
$$\max_{\tau} U\Big(\Gamma(\theta),\tau\Big) = \int_{\theta} \Gamma(\theta) u\Big(\mathbf{c}^*(\theta,\tau)\Big) d\theta$$

$$\text{s.t. } R\Big(\Gamma(\theta),\tau\Big) = \int_{\theta} \Gamma(\theta) r(\mathbf{c}^*(\theta,\tau)) d\theta \geq \bar{R}$$

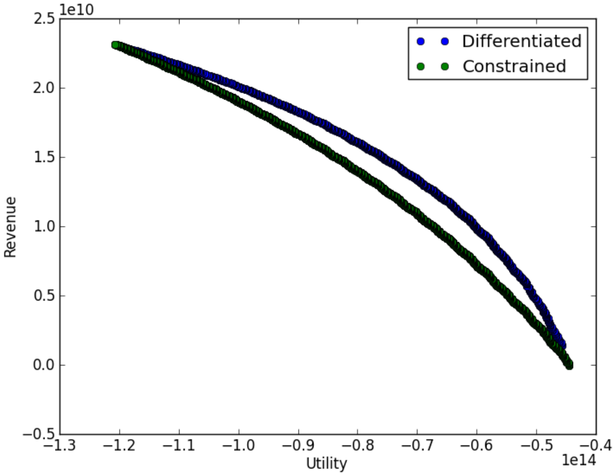
Expansion of the problem

- Solution is a point in policy space τ^*
- What if we wanted to know τ^* for all possible $\bar{R}$?

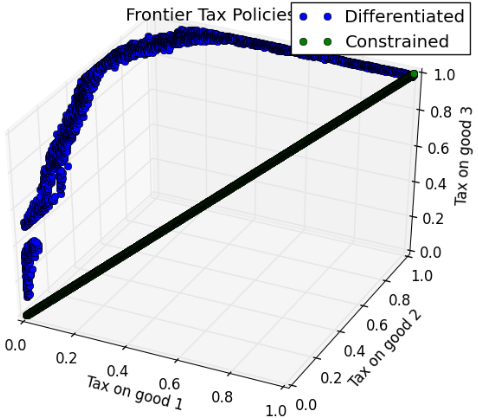
Examples of Results



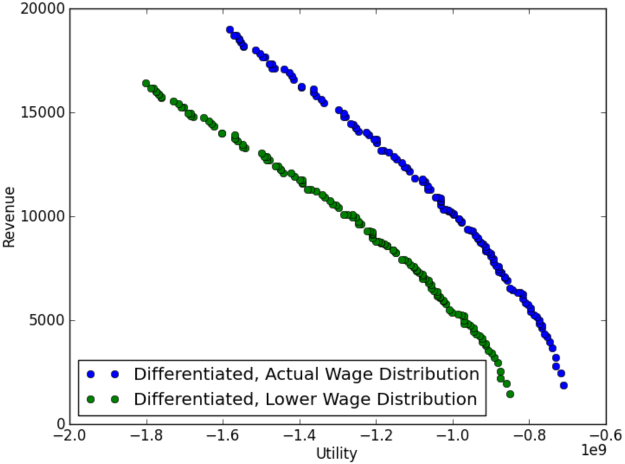
Examples of Results



Examples of Results



Examples of Results



`econsim` library

`econsim` is a Python library that we developed to solve models of the form described here. `econsim` requires only three things.

- Define the type-space and policy-space.
- Provide the model file that will define the optimization routine. It takes in a point in type-space and policy-space and returns individual utility and revenue generated.
- Provide a distribution over the type-space.

econsim nuts and bolts

- **Create Database of Individual Responses**

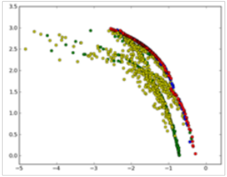
Compute individual utility and revenue for each individual and each policy..

Model File → Individual Response Database .hdf5

SLOW: DO ONCE
Creating the database is a fixed cost. We only need to create it once.

- **Find Efficient Policies Given a Distribution**

Individual Response Database .hdf5 + Distribution →



FAST: CAN DO MANY TIMES
Monte-carlo integration only requires computation of weighted sum. With modern "big-data" technology, this is fast for even large datasets.

Overview of Implementation Details

List of the most important implementation details that we will cover here.

- Big Data with HDF5
- Quasi-Monte Carlo Integration and Equidistributed Sequences (three benefits from this: coverage, refinement management, and parallelization ease)
- Multiobjective Programming: Linesweep to find Pareto efficient policies

Big Data with HDF5

What is HDF5 and why is it important for us?
HDF5 is important for two reasons:

1. It allows for efficient storage of high dimensional data that is to be accessed contiguously. (As opposed to your typical relational database.)
2. It has the ability to do parallel I/O. When dealing with extremely large files (TBs in size), I/O will be bottlenecked by the read and write speeds of the disk on which the data is stored.

Quasi-Monte Carlo Integration: Coverage

Quasi-Monte Carlo integration is used to integrate over the type space for each point in policy space given the type space distribution.

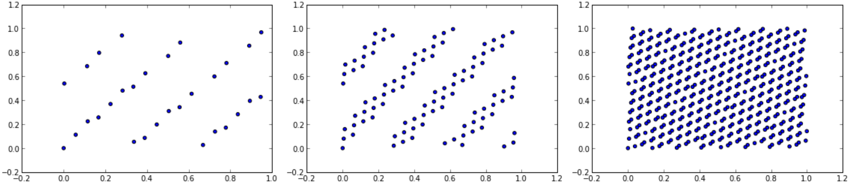
- Same as Monte Carlo Integration, but uses equidistributed sequences instead of pseudorandom numbers.
- Faster rate of convergence for large N: for s dimensions, $O\left(\frac{(\log N)^s}{N}\right)$ as opposed to $O\left(\frac{1}{\sqrt{N}}\right)$.

Quasi-Monte Carlo Integration: Coverage

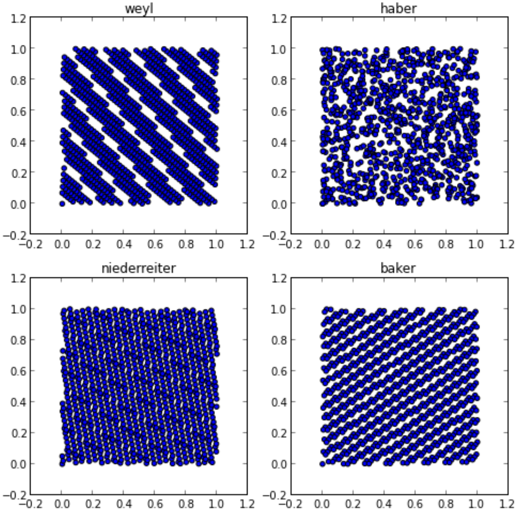
Equdistributed Sequences are deterministic sequences of real numbers where the proportion of terms falling in a subinterval is proportional to the length of that interval.

- Example: Two-dimensional Baker Sequence on interval $[0,1]$. For p_1, p_2 any rational, distinct numbers, the n^{th} item in the sequence is given by

$$\left(ne^{p_1} - \lfloor ne^{p_1} \rfloor, ne^{p_2} - \lfloor ne^{p_2} \rfloor\right)$$



Quasi-Monte Carlo Integration: Coverage



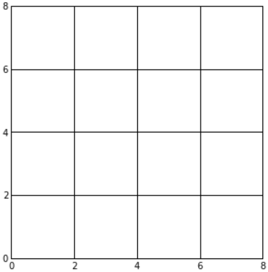
Equidistributed Sequences: Refinement Management

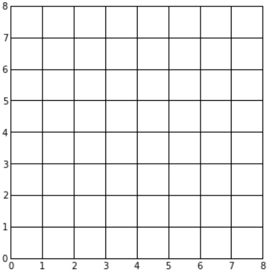
Adding resolution requires us to re-index our database of responses. Recomputing points defeats the purpose of the database.

- **Difficult:** This is difficult when we are trying to generalize the management process, especially if we would like our program to allow arbitrary numbers of dimensions in type- and policy-space.
- **Expensive:** This is costly when the database is TBs large.

Equidistributed Sequences: Refinement Management

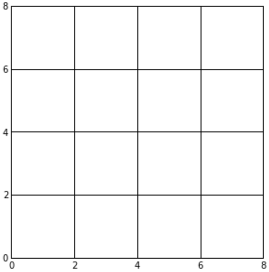
- Adding more resolution to a populated grid can lead to indexing issues.
- Consider this grid and imagine that we computed a household's response to a policy at each intersection.

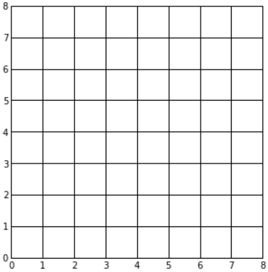




Equidistributed Sequences: Refinement Management

- Suppose to the left we have a 5x5 grid of responses.
- If we wanted more resolution later, we might want to enhance the grid to something like the 9x9 grid to the right.





Equidistributed Sequences: Refinement Management

- Using equidistributed sequences to cover the policy- and type-space makes generalizing each space to an arbitrary number of dimensions easy. Each database will always have the same shape: $N_t \times N_p \times 2$.
- Each particular point in type space or policy space can be recovered by its index. For example, in 2 dimension on the space $[0, 1] \times [0, 1]$,

```
>>> equidistributed(432, dim=2, type='weyl')
[0.9402589451771064, 0.2459488697548977l]
```

Equidistributed Sequences: Parallelization

In addition to the difficulty in resizing the response database while maintaining generality of the spaces, parallelization becomes difficult as it must choose multiple dimensions over which to slice the space.

Equidistributed Sequences: Parallelization

Divide the data among p processors.

Example 1: Consider an $N \times 1$ vector of data. This is easy to divide up.

Example 2: Consider an $N \times M$ matrix of data. You can either divide along one dimension if that dimension is large enough, or you can divide into blocks.

Example 3: Consider a higher dimensional array with a more complex shape: $N_\eta \times N_w \times N_{g_1} \times N_{g_2} \times N_{g_3}$.

Equidistributed Sequences: Parallelization

The problem is that, when the dimensionality grows, this database could be very large yet still have some dimensions that are relatively coarse. This would require us to automate the decision of how to slice up this array.

Example 3: Consider a higher dimensional array with a more complex shape: $N_\eta \times N_w \times N_{g_1} \times N_{g_2} \times N_{g_3}$.

Equidistributed Sequences: Parallelization

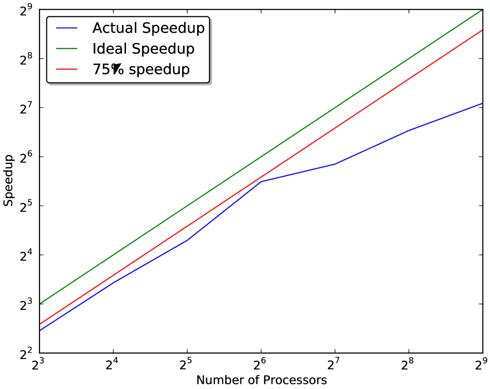
Solution: If we use equidistributed sequences, we can maintain the generality (arbitrary number of dimensions in either space) *and* keep the shape of the data square like in Example 2,

$$N_{type} \times N_{policies}.$$

Example 2: Consider an $N \times M$ matrix of data. You can either divide along one dimension if that dimension is large enough, or you can divide into blocks.

Performance: Total

Figure : Scalability of creating and refining a typical database.



Performance: Linesweep

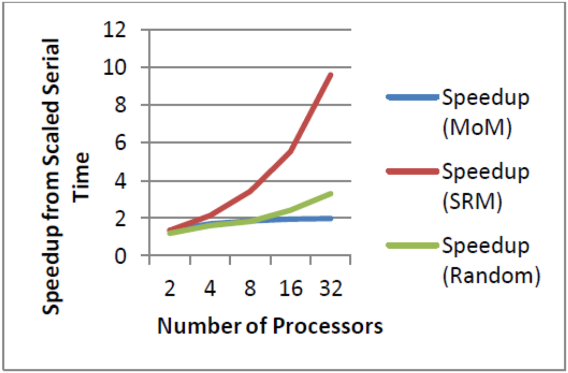
Figure : Scalability of sorting (parallel quicksort).

	n/p	100000		Unscaled Serial Time		0.013961
	p processors	2	4	8	16	32
	Predicted Time	0.019691373	0.0393794	0.059067475	0.07875553	0.0984436
	Scaled Serial Time	0.029275	0.062692	0.128646	0.289046	0.707244
Median of medians	Parallel Time	0.021526	0.036751	0.069143	0.149043	0.355726
	Scaled Speedup	1.359983276	1.7058583	1.8605788	1.93934636	1.9881707
Subcube's Root's Median	Parallel Time	0.021551	0.029298	0.03754	0.052434	0.073705
	Scaled Speedup	1.358405642	2.1398048	3.426904635	5.51256818	9.5956041
Random	Parallel Time	0.024088	0.039011	0.069904	0.119119	0.213829
	Scaled Speedup	1.215335437	1.6070339	1.840323873	2.42653145	3.3075214

	n/p	10000		Unscaled Serial Time		0.001207
	p processors	2	4	8	16	32
	Predicted Time	1.97E-02	0.0393794	0.059067475	0.07875553	0.0984436
	Scaled Serial Time	0.002508	0.005325	0.011145	0.024987	0.062274
Median of medians	Parallel Time	0.001895	0.003159	0.005931	0.01257	0.029646
	Scaled Speedup	1.32348285	1.68566	1.879109762	1.98782816	2.1005869
Subcube's Root's Median	Parallel Time	0.001897	0.002585	0.003291	0.004479	0.006205
	Scaled Speedup	1.322087507	2.0599613	3.38650866	5.5787006	10.0361
Random	Parallel Time	0.002216	0.004064	0.005728	0.011477	0.018046
	Scaled Speedup	1.131768953	1.3102854	1.945705307	2.17713688	3.4508478

Performance: Linesweep

Figure : Scalability of sorting (parallel quicksort).



Conclusion

- We describe big data solution method for big theory
  - Essential for big heterogeneity and nonconvex or discontinuous constrained optimization
- Reusable Python Package for optimal policy problems given a heterogeneous population
- Scalable parallelization in computation and data access
- Collaboration