Economics Bulletin

Volume 42, Issue 4

SpMV approaches to dynamic discrete choice models with limited transition

Yu Wang
Torotno Metropolitan University

Yao Luo
University of Toronto

Abstract

Dynamic optimization problems often involve continuous state variables. Casting such problems into dynamic discrete choice models usually requires variable discretization. When there are multiple state variables, many discretized future states will be visited with only very small probability conditional on current states. We investigate pruning these small transition probabilities and applying the sparse matrix-vector multiplication method in value function iterations. We assess our method in a numerical example inspired by Rust (1987) and Barwick and Pathak (2015). Our method substantially improves computational performance and reduces memory requirements with little loss in accuracy.

Citation: Yu Wang and Yao Luo, (2022) "SpMV approaches to dynamic discrete choice models with limited transition", *Economics Bulletin*, Volume 42, Issue 4, pages 2171-2183
Contact: Yu Wang - wangyu5@ryerson.ca, Yao Luo - yao.luo@utoronto.ca.
Submitted: August 31, 2022. **Published:** December 30, 2022.

The dynamic discrete choice model is the workhorse for modeling forward-looking behaviors in structural economics and has been widely used in empirical IO, labor, and macroeconomics. Yet, many forward-looking decisions are made based on continuous state variables, such as mileage, assets, or technology shock. Formulating such optimization problems into dynamic discrete choice models requires researchers to discretize continuous state variables. When grids are fine or when there are multiple variables, the full representation of the transition matrix can be very large due to the curse of dimensionality, rendering standard solution methods such as value function iteration infeasible.

The full representation of the transition matrix is inefficient when there are multiple state variables, and thus many future states may only be realized with very small probability. We investigate approximating these small transition probabilities as zero, representing the transition matrix in sparse format, and adopting *sparse matrix-vector multiplication* (SpMV) in value function iteration. We assess the efficiency gain of two sparse representations, *Compressed Sparse Row* (CSR) and *Block Compressed Sparse Row* (BSR), from the perspectives of both computation and memory requirements. Our

[*]Luo: Department of Economics, University of Toronto, Toronto, ON M5S 3G7, Canada. Email: yao.luo@ utoronto.ca. Wang: Department of Economics, Toronto Metropolitan University, Toronto, ON M5B 2K3, Canada. Email: wangyu5@ryerson.ca.

numerical context is a close analogue of Rust (1987) coupled with multiple continuous exogenous state variables as in Barwick and Pathak (2015).

While a fair amount of work has investigated accelerating the computation of value function iterations, much less attention has been given to incorporating SpMV. Arcidiacono et al. (2016) and Sargent and Stachurski (2022) are among the few papers and notes that mention the potential speed gain of representing a given transition matrix in sparse format.

Our approach is related to Gordon (2021), which proposes pruning low-probability grids in discretizing a VAR process. We differ by assessing the pruning of low transition probabilities, as well as applying SpMV techniques to these pruned transition matrices when solving dynamic optimization problems.

2 Dynamic Discrete Choice Models

Similar to Rust (1987), an agent makes decision $d_t \in \{0,1\}$ on whether to replace a bus engine in each time period t. Denote $d_t = 1$ for replacing the engine and $d_t = 0$ otherwise. The endogenous state variable, mileage $x_{0,t}$, evolves as follows:

$$P\big(x_{0,t+1} \big| x_{0,t}, d_t\big) = \begin{cases} \theta_0 \exp\{-\theta_0\big(x_{0,t+1} - x_{0,t}\big)\} & \text{if } d_t = 0 \text{ and } x_{0,t+1} \geq x_{0,t} \\ \theta_0 \exp\{-\theta_0 x_{0,t+1}\} & \text{if } d_t = 1 \text{ and } x_{0,t+1} \geq 0 \\ 0 & \text{otherwise} \end{cases}. \tag{1}$$

Similar to Barwick and Pathak (2015), there are several exogenous state variables capturing the aggregate market conditions. Assume there are three such variables, $x_{1,t}, x_{2,t}, x_{3,t}$, all independent from $x_{0,t}$. The exogenous state variables evolve with a vector autoregressive process,

$$\begin{bmatrix} x_{1,t+1} \\ x_{2,t+1} \\ x_{3,t+1} \end{bmatrix} = \begin{bmatrix} \theta_1 & \theta_2 & \theta_2 \\ \theta_2 & \theta_1 & \theta_2 \\ \theta_2 & \theta_2 & \theta_1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \end{bmatrix} + \mathbf{e_t}, \tag{2}$$

where $\mathbf{e_t}$ follows a multivariate normal distribution $N(0, \Sigma)$.

Denote $\mathbf{x}_t = \{x_{0t}, x_{1t}, x_{2t}, x_{3t}\}$. The flow payoff in period t is $u(\mathbf{x}_t, d_t) + \epsilon_t$, consisting of a reward function $u(\mathbf{x}_t, d_t)$ and a choice-specific preference shock ϵ_t. The flow payoff, discounted at rate $\beta \in (0,1)$ intertemporally, is specified as follows:

$$u(\mathbf{x_t}, d_t) = \begin{cases} \dfrac{\alpha_4 x_{0,t} \exp(\alpha_1 x_{1,t} + \alpha_2 x_{2,t} + \alpha_3 x_{3,t})}{1 + \exp(\alpha_1 x_{1,t} + \alpha_2 x_{2,t} + \alpha_3 x_{3,t})} & \text{if } d_t = 0 \\ \alpha_5 & \text{if } d_t = 1 \end{cases}.$$

The agent chooses d_t to sequentially maximize the discounted sum of expected payoffs:

$$E\Bigg\{ \sum_{t=1}^{\infty} \beta^{t-1} d_t \Big[u(\mathbf{x}_t, d_t) + \epsilon_t \Big] \Bigg\},$$

where the expectation in period t is taken over the future values of state variables and preference shocks.

Assuming the preference shock ϵ_t is realized in the beginning of each period t and follows a Type-I Extremum Value distribution, the standard solution of the problem is to discretize the continuous state variable vector $\mathbf{x}$ into S grids, $\mathbf{\Omega} = \{\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^S\}$, and to solve for the ex-ante value function $\overline{V}(\mathbf{x})$ of the Bellman equation for each $\mathbf{x} \in \Omega$,

$$\overline{V}(\mathbf{x}) = \log\left[\sum_{d\in\{0,1\}}\left(\exp\left(u(\mathbf{x},d) + \beta\sum_{\mathbf{x}'\in\Omega}\overline{V}(\mathbf{x})Prob(\mathbf{x}'|\mathbf{x},d)\right)\right)\right].$$

The Bellman equation can be cast in matrix representation:

$$\overline{\mathbf{V}} = \log\left[\sum_{d\in\{0,1\}}\left(\exp\left(\mathbf{u}(d) + \beta\mathbf{Q}(d)\overline{\mathbf{V}}\right)\right)\right], \tag{3}$$

with the i-th element of $\mathbf{u}$ being $u(\mathbf{x}^i, d)$, and the (i,j)-th element of matrix $\mathbf{Q}(d)$ being $Prob(\mathbf{x}^j|\mathbf{x}^i, d)$. It can be proven that equation (3) consists of one unique fixed point. The standard method to find this unique solution $\overline{\mathbf{V}}$ is *value function iteration*, where researchers start with an initial guess $\overline{\mathbf{V}}^0$, then sequentially apply

$$\overline{\mathbf{V}}^{k+1} = \log\left[\sum_{d\in D}\left(\exp\left(\mathbf{u}(d) + \beta\mathbf{Q}(d)\overline{\mathbf{V}}^k\right)\right)\right]$$

until $\overline{\mathbf{V}}^{k+1} \approx \overline{\mathbf{V}}^k$.

3 Applying SpMV to Value Function Iterations

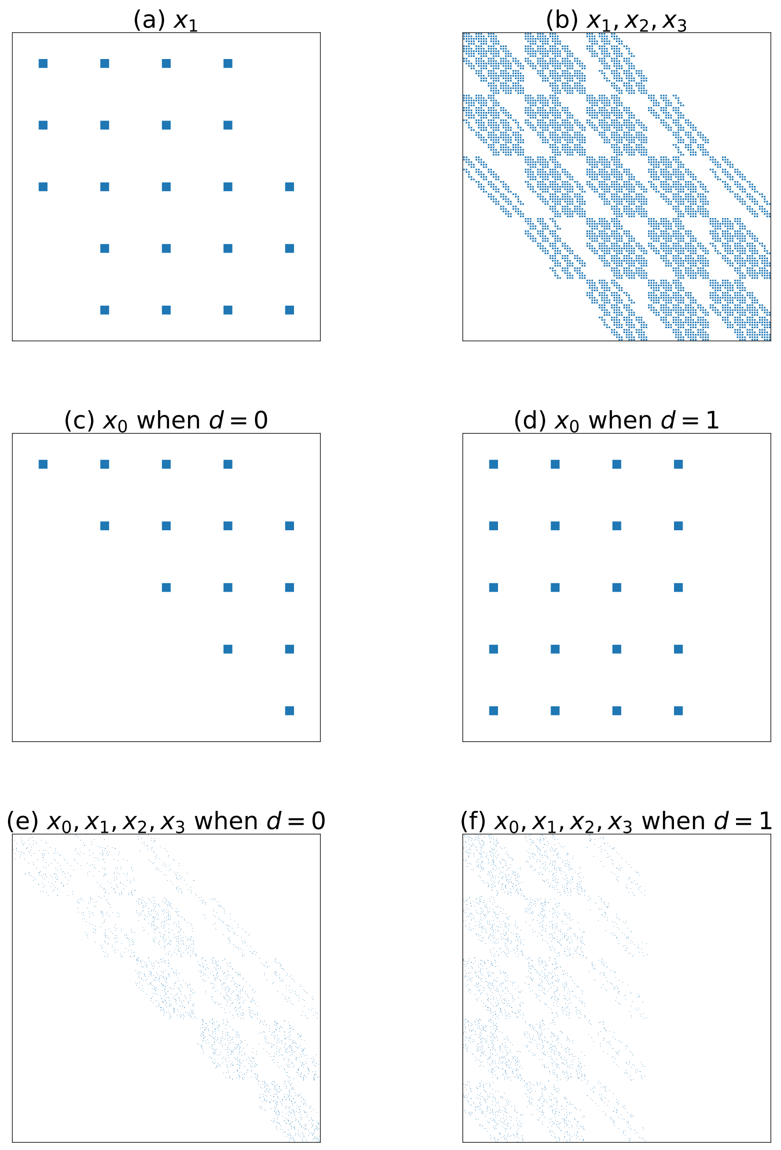
To see how costly the standard matrix-vector multiplication $\mathbf{Q}(d)\overline{\mathbf{V}}$ can be, note that the size of the transition matrix $\mathbf{Q}$ grows quickly with the number of grids and state variables. Assuming $\mathbf{X}$ has N dimensions, with each dimension discretized into K grids, the size of $\mathbf{\Omega}$ will be $S = K^N$, and the size of the transition matrix will be $S^2 = K^{2N}$. The computational complexity of calculating $\mathbf{Q}(d)\overline{\mathbf{V}}$ once is $O(K^{2N})$ for each action d. Consider the empirical model in Barwick and Pathak (2015), storing the transition matrix with 20 grid points for each of four state variables requires over 190 GB of RAM.

On the other hand, the number of state variables also contributes to the sparsity of the transition matrix, as the joint probability of multiple individually low-probability states occurring simultaneously can be extremely small. For instance, consider a N-dimensional random vector Y following $\mathcal{N}(0, \mathbf{I})$. The probability $Prob(Y_n \leq -2.5)$ is roughly 0.006 for each dimension n, which is small but still non-negligible. Yet the joint probability $Prob(Y_n \leq -3)$ for all n is around 0.006^N and becomes a tiny $2.1e^{-7}$ when $N = 3$.

Figure 1 panels (a) and (b) illustrate how this force drives the sparsity in the transition matrix of our numerical example. Each one of the four state variables is discretized into five grids. Each dot represents a transition matrix element greater than $5.0e^{-4}$. The transition matrix of a single AR(1) variable, as shown in Panel (a), has only 16% of elements below the cutoff. Yet, the transition matrix of a VAR(1) vector of dimension 3 with each dimension following the same AR(1) process in Panel (a) has 65% of elements below the cutoff, as shown in Panel (b). Additionally, Panel (c) of Figure 1 illustrates that the future mileage can only increase when not replacing the engine.

Following the discussion above, we investigate whether approximating these small

Figure 1: Sparsity Patterns in Transition Matrices



Note: x_0 represents the endogenous state variable, mileage; x_1, x_2, and x_3 represent the three exogenous state variables following a VAR distribution; $d = 0$ represents not replacing the engine and $d = 1$ represents otherwise. Each of the four state variables is discretized into 5 grids; $\theta_0 = 2.0$, $\theta_1 = 0.75$, $\theta_2 = 0.0$. $\Sigma = I$. Each blue dot represents a matrix element greater than $5.0e^{-4}$.

transition probabilities as zero, rescaling the remaining probabilities such that each row of $Q(d)$ still adds up to 1, and storing the rescaled transition matrices in sparse format can increase computational performance and reduce memory requirement. We first consider the most common sparse format, CSR. CSR stores in memory only the nonzero elements and their positions, and multiplies these non-zero elements with the correspondingly-positioned elements in the vector $\overline{\mathbf{V}}$. Since this approach skips the data movement and operation of the zero elements, it can substantially speed up the computation process when sparsity is high. Specifically, consider an $m \times n$ matrix with NNZ non-zero elements, represented in CSR format with three one-dimensional arrays $(\mathbf{a}, \mathbf{ia}, \mathbf{ja})$. The arrays $\mathbf{a}$ and $\mathbf{ja}$ are of length NNZ, and contain the non-zero values and column indices of those values, respectively. The array $\mathbf{ia}$ is of length $m+1$, with $\mathbf{ia}[1] = 1$ and $\mathbf{ia}[i+1]$ encoding 1 plus the total number of non-zeros above row i. See Appendix A.1 for more details and an example.

The second sparse format we investigate is BSR. This is motivated by the observation that nonzero elements in transition matrices are often contiguous. For instance, consider $y_{t+1} = \phi_0 + \phi_1 y_t + \varepsilon_t$, with $\varepsilon_t \sim N(0, \sigma^2)$. Given current state y_t, the future states that will be realized with non-negligible probabilities are all surrounding $\phi_0 + \phi_1 y_t$ and thus form a dense block.[1] BSR partitions the transition matrix using $c \times c$ submatrices and, instead of storing the position of each nonzero element, stores the position of each submatrix containing at least one non-zero element. Thus, when iterating over all nonzero elements of a matrix and multiplying them with the vector, the SpMV algorithm takes out a $c \times c$ block of nonzero elements each time it visits the memory under BSR, rather than a single element as under CSR. This reduction in frequency with which SpMV accesses the memory under BSR may further speed up the computation process.[2]

Last, we discuss how sparse representation reduces the memory requirement to store the transition matrix. As a back-of-the-envelope calculation, each element of an array requires eight bytes of RAM if using double-digit precision or four bytes of RAM for integers. Therefore, an $(S \times S)$-sized dense double-digit-precision matrix occupies $S^2 \times 8/1024^3$ gigabytes of RAM. The storage in CSR is the sum of storage for three arrays, $\big(NNZ \times 8 + (S+1) \times 4 + NNZ \times 4\big)/1024^3 = \big(NNZ \times 12 + (S+1) \times 4\big)/1024^3$ gigabytes, which is much smaller when NNZ is small. The storage in BSR is similar to that of CSR. Continuing the example in Barwick and Pathak (2015) with four variables discretized into 20 grids each, if 10% of the elements are nonzero, CSR takes up only 28.6GB, which is well within the RAM capacities of most modern CPUs.

Many software support converting a matrix into its sparse representation. A practical concern, however, is that researchers may not have sufficient computer RAM to construct the full transition matrix and supply it to the software in the first place. However, it is often feasible to construct the transition matrix element-by-element and fill $(\mathbf{a}, \mathbf{ia}, \mathbf{ja})$ correspondingly along the way.

One such case is when state variables can be divided into several groups and are independent across groups (though not necessarily within group). Researchers can first construct the transition matrix for each group, and compute the full transition matrix as the Kronecker product of these transition submatrices. Thus, researchers would only need sufficient RAM to support these transition submatrices instead of the full transition matrix. In our numerical example, $x_{0,t}$ and $\{x_{1,t}, x_{2,t}, x_{3,t}\}$ are independent. Considering

[1]This pattern can be seen in Panel (a) of Figure 1. When there are multiple state variables, the transition matrix still contains many dense blocks, as shown in Panel (b),(e), and (f) of Figure 1.

[2]See Vuduc (2003) for an in-depth discussion of BSR.

20 grids of each variable, the size of these two submatrices combined is only $\big((20^3)^2 + 20^2\big) \times 8/1024^3$ =0.48GB.

The second case is when all state variables follow a VAR distribution. Tauchen (1986) proposes linearly transforming the VAR process and discretizing each dimension of the transformed VAR. Following this method, we are able to compute the value of each element in the transition matrix, knowing only its row and column index.

4 Results

To evaluate the accuracy, memory requirements, and run time of our proposed method, we solve our numerical example under various numbers of grids and parameter value specifications. Specifically, we implement three solvers using Fortran 90 on a Linux terminal with 8GB RAM. The *Standard* Solver is the standard value function iteration with full representation of the transition matrix. The *CSR* and *BSR* Solvers adopt the corresponding sparse representations. We discretize each of the four state variables into K grids, where $K = \{3, 4, ..., 12\}$. We stop at 12 as we need to compare the accuracy of SpMV against the standard approach, and 12^4 is the largest matrix dimension we can accommodate. See Appendix AC for more technical details.

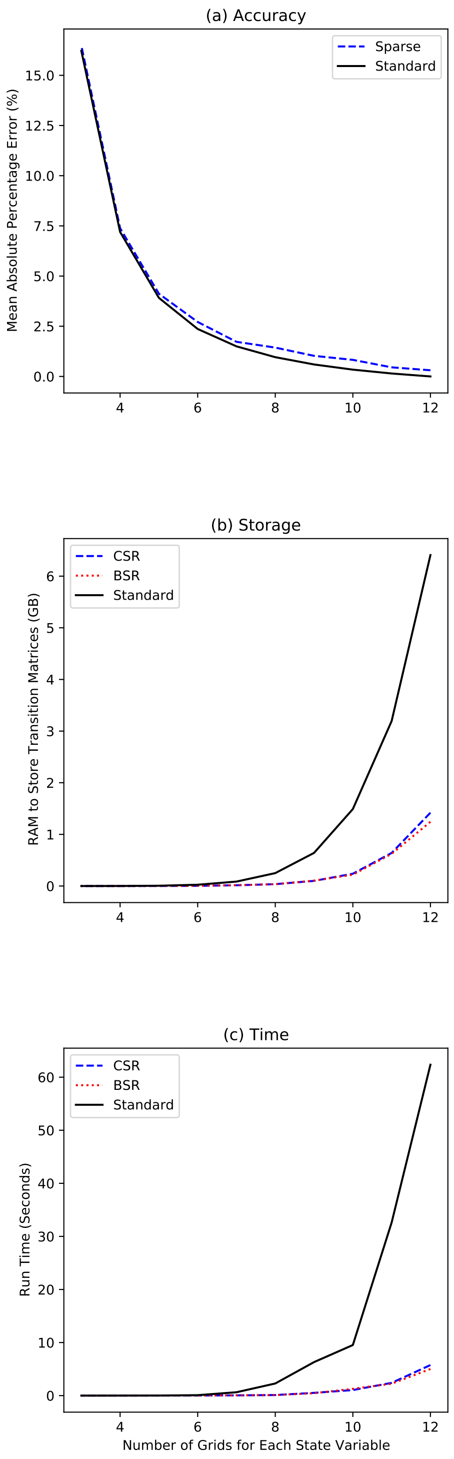
Figure 2 summarizes the results as state variables are discretized into finer grids. Our benchmark value function is the solution of the *Standard* solver with $K = 12$. As shown in Panel (a), the mean absolute percentage errors drop quickly with the number of grids, suggesting a large difference between estimates under coarser and finer grids. The errors are almost identical for both standard and sparse solvers, suggesting that the sparse and standard solvers generate sufficiently comparable results. Panel (b) shows that the computational time for *Standard* solvers remains similar to sparse solvers until $K = 6$, and grows to 10.8 and 12.4 times of *CSR* and *BSR* solvers when $K = 12$. Similarly, as shown in Panel (c), the memory requirements are similar for all solvers until $K = 6$, while the *Standard* solver takes up 5.1 and 4.5 times the storage space as the *CSR* and *BSR* solvers, respectively, when $K = 12$.

We then evaluate the three solvers when fixing $K = 12$ and varying the parameters θ_1 and θ_2 in equation (2). As shown in Appendix A.2, the mean absolute percentage errors remain below 0.4% in all our specifications. The speed gains and the reduction in memory requirement of sparse solvers relative to the *Standard* solver is also robust and stable throughout. The correlation between each exogenous state variable and its own lagged value, θ_1, appears to have a large effect on the storage requirement and computational time of sparse solvers, as shown in Figure 3 Panel (a). As we increase θ_1 from 0.05 to 0.95, storage requirements decline 40% for *CSR* and 36% for *BSR*, and computational time declines 33% for both *CSR* and *BSR*. Intuitively, high autocorrelation θ_1 increases the sparsity, as realizations of future values far away from the lagged values are even less likely to occur. The correlation between each exogenous state variable and other variables' lagged values, θ_2, has little impact on the performance of sparse solvers.

5 Conclusion

This paper investigates pruning the low transition probabilities when discretizing continuous state variables in dynamic discrete choice models and applying sparse matrix-vector representation in value function iteration. Our numerical example suggests a substantial

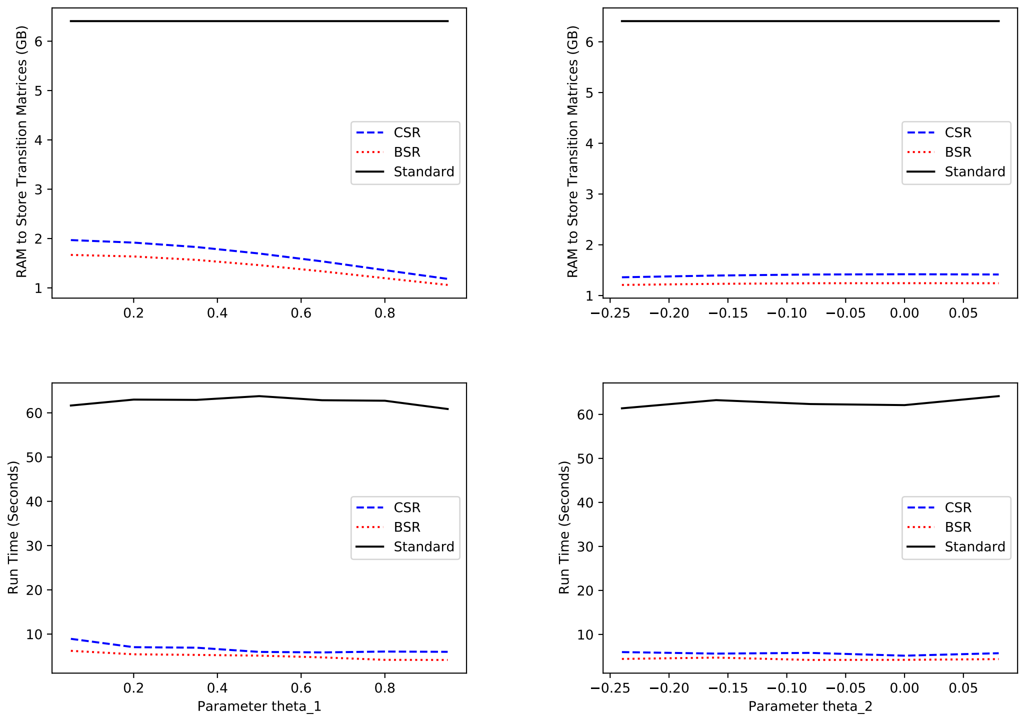
Figure 2: Comparison of Solvers Across Numbers of Grids



Note: $\theta_0 = 2.0$, $\theta_1 = 0.75$, $\theta_2 = 0.0$. $\Sigma = I$.

Figure 3: Comparison of Solvers with Different Parameter Values

(a) Varying θ_1 (b) Varying θ_2



Note: $K = 12$, $\theta_0 = 2.0$, $\theta_1 = 0.75$, $\theta_2 = 0.0$. $\Sigma = I$. Each panel varies one parameter while holding constant all other parameters.

speed gain and reduction in memory requirements with very small loss in accuracy. The method proposed in the paper should significantly expand the ability of researchers to solve and estimate dynamic discrete choice models. Lastly, although interesting and important, studying how SpMV approaches affect parameter estimates is beyond the scope of this article. Since Rust (1987), several estimation methods for standard dynamic discrete choice models have become available. See Aguirregabiria and Mira (2010) for a survey of methodologies and Luo and Sang (2022) for penalized sieve estimators.

References

Aguirregabiria, Victor, and Pedro Mira. 2010. "Dynamic discrete choice structural models: A survey." *Journal of Econometrics*, 156(1): 38–67.

Arcidiacono, Peter, Bayer Patrick, Blevins Jason, and Ellickson Paul. 2016. "Estimation of Dynamic Discrete Choice Models in Continuous Time with An Application to Retail Competition." *Review of Economic Studies*, 83(3): 889–931.

Barwick, Panle Jia, and Parag A. Pathak. 2015. "The Costs of Free Entry: An Empirical Study of Real Estate Agents in Greater Boston." *The Rand Journal of Economics*, 46(1): 103–145.

Gordon, Grey. 2021. "Efficient VAR Discretization." *Economics Letters*, 204(109872).

Luo, Yao, and Peijun Sang. 2022. "Penalized Sieve Estimation of Structural Models." *Working Paper.*

Rust, John. 1987. "Optimal Replacement of GMC Bus Engines: An Empirical Model of Harold Zurcher." *Econometrica*, 55(5): 999–1033.

Sargent, Thomas J, and John Stachurski. 2022. "Discrete State Dynamic Programming."

Tauchen, George. 1986. "Finite State Markov-Chain Approximations to Univariate and Vector Autoregressions." *Economics Letters*, 20: 177–181.

Vuduc, Richard Wilson. 2003. "Automatic Performance Tuning of Sparse Matrix Kernels." PhD diss. University of California, Berkeley.

A Appendix

A.1 Compressed Sparse Rows (CSR)

Algorithm 1 summarizes the multiplication process given $\mathbf{Q}$ in CSR format. Since this algorithm operates solely on the nonzero elements, the time complexity is reduced from $O(mn)$ to $O(NNZ)$. This is consistent with the indexing practice of Fortran and Matlab. We stick to this one-based indexing practice throughout this paper. Alternatively, there is *zero-based indexing* corresponding to C, C++, and Python, with **ia** starting from index 0 rather than 1, $\mathbf{ia}[0] = 0$, and $\mathbf{ia}[i]$ encoding the total number of non-zeros above row i.

Algorithm 1 SpMV Algorithm with CSR Representation

```
1: procedure SpMV_CSR($\mathbf{a}, \mathbf{ia}, \mathbf{ja}, \overline{\mathbf{V}}$)          ▷ $\mathbf{Q}$ is represented by $\mathbf{a}$, $\mathbf{ia}$, $\mathbf{ja}$
2:     for $i = 1, length(\mathbf{ia}) - 1$ do                           ▷ Iterate over rows of $\mathbf{Q}$
3:         $yc = 0$                                                ▷ Scalar replacement since reused
4:         for $k = \mathbf{ia}(i), \mathbf{ia}(i+1) - 1$ do              ▷ Iterate over nonzero entries in row $i$
5:             $yc \leftarrow yc + \mathbf{a}(k) \times \overline{V}(\mathbf{ja}(k))$       ▷ Element-wise multiplication
6:         $y(i) = yc$
7:     return $y$                                                  ▷ The output array
```

For instance, consider the following transition matrix:

$$\mathbf{Q} = \begin{bmatrix} 0.7 & 0.3 & 0 \\ 0.1 & 0.8 & 0.1 \\ 0 & 0.1 & 0.8 \end{bmatrix}. \tag{4}$$

The CSR representation of $\mathbf{Q}$ is shown in Table I. The SpMV multiplies each element of $\mathbf{a}$ with the corresponding element in $\overline{\mathbf{V}}$, sums the products by each row of $\mathbf{Q}$, and generates the output array.

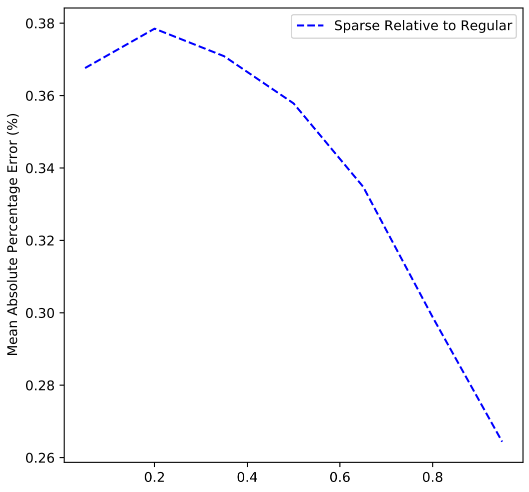
Table I: Example: CSR Representation for $\mathbf{Q}$ in Equation (4)

a	(0.7	0.3	0.1	0.8	0.1	0.1	0.8)
ia	(1	3	6	9)			
ja	(1	2	1	2	3	2	3)

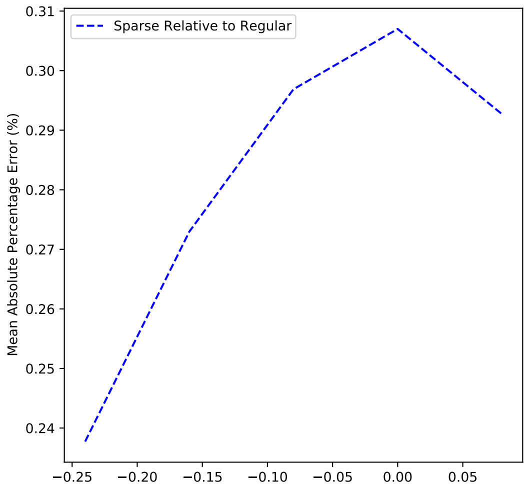
A.2 Accuracies of SpMV

Figure 4: Comparison of Accuracies of Solvers with Different Parameter Values

(a) Varying θ_1



(b) Varying θ_2



Note: $K = 12$, $\theta_0 = 2.0$, $\theta_1 = 0.75$, $\theta_2 = 0.0$. $\Sigma = I$. Each panel varies one parameter while holding constant all other parameters.

Specifically, we implement three solvers using Fortran90 on a Linux terminal with Intel® Xeon® CPU E5-2683 v4 @ 2.10GHZ and 8GB RAM. The *Standard* Solver is the standard value function iteration with full representation of the transition matrix. It uses the highly-tuned dense matrix-vector multiplication operator *dgemv* from the *Intel OneAPI Math Kernel Library*. The *CSR* Solver employs the SpMV operator *mkl_dcsrgemv* for CSR representation from the same library. The *BSR* Solver converts the CSR representation to a BSR representation using operator *mkl_dcsrbsr* and then employs the SpMV operator *mkl_dbsrgemv*. In BSR, we partition the transition matrices using 2×2 submatrices.

When discretizing state variables, we stop at 12 because 12^4 is the largest matrix dimension we can accommodate. We have two transition matrices corresponding to $d = 0$ and $d = 1$. $K = 12$ generates two matrices taking up 6.4GB combined. $K = 13$ generates two matrices taking up 12.2GB combined. We discretize the three exogenous variables following Tauchen (1986). Our precision cutoff for sparse representation is $5.0e^{-4}$ when $K \leq 6$, $2.0e^{-4}$ when $K \in \{7, 8\}$, $1.0e^{-4}$ when $K = 9$, $6.0e^{-5}$ when $K = 10$, $2.0e^{-5}$ when $K = 11$, and $1.0e^{-5}$ when $K = 12$. All elements below the precision cutoff are treated as zero. We select these cutoffs such that the sum of the transition probabilities approximated to zero remains below 2%. For all solvers, our initial guess is $\overline{\mathbf{V}}^0 = 0$. Convergence is defined as $\max |\overline{\mathbf{V}}^{\mathbf{k+1}} - \overline{\mathbf{V}}^{\mathbf{k}}| < 1.0e^{-8}$.

We measure accuracy, memory requirements and run time of the three solvers as follows. For accuracy, we first generate 5^4 benchmark grids by discretizing the four state variables. Next, we set our benchmark value function as the solution of the *Standard* solver with $K = 12$. Then, we compute the mean absolute percentage error between the value function estimates of each solver and the benchmark value function at these 5^4 benchmark grids. If the value functions in question are not evaluated at these benchmark grids (for instance, when we evaluate the value function with $K = 7$), we linearly interpolate their values. *CSR* and *BSR* generate identical value function solutions, so we plot the accuracy of *CSR* only. For memory requirements, we mean the RAM needed to store the transition matrices. For run time, we capture the computational time for the value function iteration part of each solver. We do not include the time to construct a transition matrix in either standard or sparse format. As the transition matrices remain the same throughout the value function iteration, this step only needs to be completed once and can be performed separately from the value function iteration.