Ensuring Efficient Convergence to a Given Stationary Distribution

Kenneth Peluso

September 2017 - April 2018

Abstract

How may we find a transition matrix that guarantees the long-run convergence of a Markov Chain to a given stationary distribution? Solving for this (usually) undetermined system is non-trivial and presents unique computational challenges. We begin to tackle this issue by reviewing Markov Chains and related definitions. Eight different of methods of directly solving for a transition matrix are presented along with their limitations. Relaxations of the two assumptions - the Identityless and Independence Assumptions - underlying these direct methods are considered. Two methods of generating a Mass Matrix - the transition matrix underlying hops between entire population states - are described while developing the notion of successively-bounded weak compositions and describing an algorithm for their exhaustive generation. Applications of some methods are provided with respect to optimizing firm profit via optimally distributing workers among wage bracket and optimizing measures of national wealth via manipulation of class distribution, economic inequality and immigration policy.

1 Introduction

Given a set of finite bins, a finite population to be allocated among the bins, and an objective function that is optimized on some set of population distributions across the bins, conventional optimization practices fall short of pragmatic. Integer programming and other methods may yield an optimal population distribution, but how may we guarantee that the population converges to such an optimal distribution? This paper attempts to answer that question.

At least one optimal distribution is assumed to exist and be calculated. One of these distributions is selected and normalized to 1. We assert that this distribution, a discrete probability distribution, is a stationary distribution for some Markov Chain. This Markov Chain is assumed to provide a complete, probabilistic description of the evolution of this population. Our goal is to define that Markov Chain. In particular, we seek to find a transition matrix, the *Individual Matrix*, whose first eigenvector (i.e. the eigenvector corresponding to an eigenvalue of 1) is the asserted stationary distribution.

For $n > 2$, this is an underdetermined problem - there exist far more variables than constraints. These variables are the elements of the desired transition matrix. The few constraints available to us consist of a system of linear equations arising from the matrix multiplication between the transition matrix and its stationary distribution, in addition to the requirement that all columns of the transition matrix must sum to 1 (i.e. that all columns of the transition matrix are probability distributions). Due to the underdetermined nature of the problem, many transition matrix solutions may exist, but finding them is nontrivial.

We begin by clearly listing all preliminary definitions, as the literature concerning Markov Chains is ripe with inconsistencies between papers. The general problem is then clearly restated along with its assumptions. Methods of directly finding a solution given these assumptions are presented. Some methods admittedly perform better than others, but all are presented regardless. Those which require no more than 2 bins are detailed first, those lacking that requirement follow. A brief aside regarding the performance of each algorithm accompanies its description and pseudocode.

As the paper continues, we relax the 2 core assumptions while providing motivation for a much more complex and tangentially related problem - that of finding the *Mass Matrix*, to be defined later. To generate this matrix, new definitions are required, in particular, we must generalize slightly beyond the restricted weak compositions of primary concern in Page 2012 [Pag12]. Algorithms comparable to those in Page 2012 are created to generate instances of this new definition, of *successively-bounded weak compositions.*

We continue by surveying some applications of our methods used to directly solve for the Individual Matrix. We discuss methods to ensure optimal convergence of...

- the distribution of citizens in a society among income brackets to maximize GDP or overall welfare.
- the distribution of employees in a firm among wage brackets to maximize the firm's profit.
- the distribution of citizens among economic classes to an ideal level of inequality in a nation.
- the genetic diversity of a nation to an ideal level to maximize income-per-capita.

For these applications, we reassert the two core assumptions – the Identityless and Independence Assumptions. The latter two applications build off of the work of Oded Galor and his colleagues, and connections are drawn between their previous work and this new material. A generalization of all possible applications concludes the application section.

The paper concludes with a summary of all that has been discussed and suggestions for future research.

2 Definitions

The literature on applied probability models is ripe with notation that is inconsistent across texts, thus it is important to agree on a common language from the very beginning.

Definition 1 (Markov Chain (MC))**.** *A stochastic process* $X = (X_i)_{i=1:k}$ *with state space* $S = 1, 2, 3, ...$ *is a Markov Chain (MC) if it satisfies the Markov Property i.e. if for all* $n \in \mathbb{N}$ *and all states* $i_1, i_2, ...i_n$ *it follows that*

$$P(X_n = i_n | X_{n-1} = i_{n-1}, X_{n-2} = i_{n-2}, ..., X_1 = i_1) = P(X_n = i_n | X_{n-1} = i_{n-1})$$

[Hua77]

In this paper, we only consider *homogeneous* MCs of discrete times, where the probability of hopping between states is fixed with respect to time.

The matrix that fully describes the MC is called the *transition matrix P*. If the state space of the MC is n, then P is an $n \times n$ matrix. p_{ij} is an element in P found at the intersection of row i and column j and is the probability that $X_t = i$ given that $X_{t-1} = j$.

Definition 2 (Irreducible MC)**.** *An MC with state space* S *is irreducible if* $\forall C \in S, \exists p_{ij} > 0 \ni i \in C, j \notin C$.

In other words, it is possible to reach one state in S from any other state in S in an irreducible MC. An MC that is not irreducible is called *reducible.*

Definition 3 (Aperiodic MC)**.** *An MC with state space* S *and transition matrix* P *is aperiodic if all states in* S *are aperiodic i.e. when* $\forall i \in S, gcd(C) = 1$ *where* $C = \{t \in \mathbb{N} | p_{ii}^{(t)} > 0\}$*,* $gcd(C)$ *is the greatest common divisor of the elements in set* C *and* $p_{ii}^{(t)}$ *is the element in matrix* P^t *at row* i*, column* i*. [Hua77]*

Loosely put, an aperiodic MC allows all states in S to be returned to be realized multiple times over as the number of time steps increases indefinitely. Furthermore, the lengths of elements in the set of possible ways to return to each state, starting from that state, vary or is 1 if there is only 1 path from one state to itself, namely the path $i \to i$ where $i \in S$.

As we will see, forcing our solution transition matrix to have a stationary distribution will force it to to be irreducible, but not necessarily aperiodic.

Definition 4 (Recurrent (Persistent)). *For a homogeneous MC, a state j is recurrent (or persistent) if*

$$\sum_{t=1}^{\infty} f_{jj}^{(t)} = 1$$

[Hua77]

In other words, if in some, perhaps infinite time, the MC can return to any state it starts at.

Definition 5 (Positive Recurrent). *For a homogeneous MC, a state j is positive recurrent if it is recurrent and if*

$$\sum_{t=1}^{\infty} t f_{jj}^{(t)} < \infty$$

Definition 6 (Null Recurrent). *For a homogeneous MC, a state j is null recurrent if it is recurrent and if*

$$\sum_{t=1}^{\infty} t f_{jj}^{(t)} = \infty$$

[Hua77]

Furthermore, in a finite, homogeneous MC, all recurrent states are positive recurrent. [Hua77]

Definition 7 (Ergodic). *Let P be the $n \times n$ transition matrix for a homogeneous MC and let π be some $n \times 1$ vector of positive entries. If $\lim_{t\to\infty} p_{ij}^{(t)} = \pi_j$ for all j independently of i and $\sum_{j=1}^{n} \pi_j = 1$ then the MC is ergodic and π is called the stationary distribution of the MC. [Hua77]*

It should be noted that the stationary distribution of an MC with transition matrix P is the first eigenvector of P, as in, it corresponds to the largest eigenvalue of P, which must be 1.

Furthermore, ergodicity exhibits the following two necessary conditions:

1. All recurrent states are aperiodic.
2. There is at most one irreducible closed subset of recurrent states.

[Hua77]

Therefore, if we assert the existence of a single stationary distribution for an MC, then that MC must be ergodic and therefore all of its recurrent states are aperiodic and there is at most one irreducible closed subset of recurrent states.

Theorem 1 (Fundamental Limit Theorem for Markov Chains). *For any irreducible, aperiodic MC with entirely positive recurrent states, then there exists a unique stationary distribution. [uI]*

There are two different twists to this theorem. If we drop the aperiodic assumption, then we'll still have a unique stationary distribution but we may not observe convergence to it. If we drop the irreducible assumption, then we will have a set of stationary distributions, and we may observe convergence to any elements of that set or convex combinations of any number of elements int that set. Therefore, when we say that or force a matrix to have at least one stationary distribution, we force it to be aperiodic, but not necessarily irreducible. Thus, our solutions may only sometimes be ergodic. This will matter later in the paper after we discuss mass MCs.

We will list one more definition for now. The following definition will arise during our discussion on mass matrices toward the end of this paper:

Definition 8 (Weak Composition)**.** *Let* $\texttt{WC}(N, n)$ *denote the set of all possible weak combinations of* N *identical balls and* n *unique bins. A Weak Combination* $w \in \texttt{WC}(N, n)$ *is any* n*-length list of non-negative integers that sums to* N*.*

[Csi]

3 Problem Definition

Given a population of N individuals, a set of n bins, each with an associated value function dependent only on the number of people in the corresponding bin (i.e. nodal functions), an overall welfare function that we wish to optimize that takes all nodal function as input (i.e. welfare function), our task is to find some homogeneous transition matrix governing how each individual hops about each bin at each time step. We'll call this the *individual transition matrix*, $\mathbb{P}^{(I)}$. For now, we assume that all individuals are non-unique - the *Identityless Assumption.* Therefore, all individuals abide by the same $\mathbb{P}^{(I)}$.

We will impose one more assumption, namely that any individual, regardless of their current bin-placement, cannot affect the movement of any other individual - the *Independence Assumption.*

Given these assumptions let us delve deeper into the meaning of $\mathbb{P}^{(I)}$. Consider the following generalized $\mathbb{P}^{(I)}$:

$$\begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{bmatrix}$$

What do each of these p_{ij} values mean? Each is the probability of any single individual hopping from bin j to bin i. We assume that every individual abides by this same $\mathbb{P}^{(I)}$ (courtesy of the Identityless Assumption) and that the movements and placements of individuals do not affect those of other individuals (courtesy of the Independence Assumption). Therefore, under these two, core assumptions, we may completely describe the evolution of our population once we obtain $\mathbb{P}^{(I)}$. We must now find a means (or multiple means) of solving for this $\mathbb{P}^{(I)}$.

4 Finding $\mathbb{P}^{(I)}$ Directly when $n = 2$

In this section, we survey multiple means of directly calculating a $\mathbb{P}^{(I)}$ when $n = 2$, namely bS, rS, and brS. Aside from working well when $n = 2$, all the following methods will become important in the Gibbs-Inspired Method, which can tolerate situations where $n > 2$.

We must first begin with a derivation, atop of which we will construct our 3 "$n = 2$"-case methods. Let us begin by setting $n = 2$. Then Equation (3) is a system of 2 linear equations with 2 unknowns, namely elements p_{11} and p_{22} of $\mathbb{P}^{(I)}$. These equations are:

$$p_{11}\pi_1 + (1 - p_{22})\pi_2 = \pi_1$$
$$(1 - p_{11})\pi_1 + p_{22}\pi_2 = \pi_2$$

first equation becomes:

$$p_{11} = 1 - (1 - p_{22})\frac{\pi_2}{\pi_1}$$

plug that into second equation:

$$(1 - 1 + (1 - p_{22})\frac{\pi_2}{\pi_1})\pi_1 + p_{22}\pi_2 = \pi_2$$

$$(1 - p_{22})\pi_2 + p_{22}\pi_2 = \pi_2$$

and we arrive at a tautology! As a result of the constraints requiring our columns to be distributions (and thus sum to 1), our first two equations were essentially the same. We can thus sample p_{22} randomly and obtain values from the following line to find a solution:

$$p_{11} = 1 - (1 - p_{22})\frac{\pi_2}{\pi_1} \tag{1}$$

but we must only accept values of $p_{11} \geq 0$ from this line, so we must impose the following constraint:

$$0 \leq p_{11} \Rightarrow p_{22} \geq 1 - \frac{\pi_1}{\pi_2} \tag{2}$$

We will rely on both Equation (1) and Constraint (2) heavily in upcoming algorithms.

4.1 Binary Search (bS) for $n = 2$ Case

4.1.1 Description

We will be taking full advantage of (1) and (2). Again, this method only applies when $n = 2$. Let $n = 2$ and let ev() be a function that inputs a matrix and outputs its eigenvector corresponding to an eigenvalue of 1. Let avg() take a list of numbers as input and output their average. We may now adapt the classic Binary Search algorithm for our purposes.

4.1.2 Algorithm

```
1. If $\pi_1 \geq \pi_2$ :
         $b_1 := 0$
   Else:
         $1 - \frac{\pi_1}{\pi_2}$
2. $b_2 := 1$ ; $iters := 2000$ ; $output := \begin{bmatrix} x_{11} & 1-p \\ 1-x_{11} & p \end{bmatrix}$
   where $p = \texttt{avg}([b_1, b_2]), x_{11} \sim U[0,1]$
3. While $output * \texttt{ev}(output) \neq \pi$ and $iters > 0$:
4.         If $\texttt{ev}(output)_2 < \pi_2$ :
5.               $b_1 := \texttt{avg}([b_1, b_2])$
6.               $output_{22}$ += $\texttt{avg}([b_1, b_2]) - b_1$
```

```
7.          Else:
8.              $b_2 \coloneqq \texttt{avg}([b_1, b_2])$
9.              $output_{22} \mathrel{-}= \texttt{avg}([b_1, b_2]) - b_1$
10.     $output_{11} \coloneqq 1 - (1 - output_{22})^{\frac{\pi_2}{\pi_1}}$
11.     $iters \mathrel{-}= 1$
12. Return $output$
```

4.1.3 Performance

The worst case cost for implementing bS is $O(F(d)log_2(d))$, where $F(d)$ is the cost of calculating with arithmetic operations with d-digit precision. It should be noted that d in practice is determined by the threshold one may use to determine if 2 floating-point numbers are equal in addition to the user's machine and programming language.

Proof. Let x is the most number of times we can bisect a number, d. We are always bisecting our search space of d in half, so:

$$1 = d/2^x \Rightarrow log_2(d) = x$$

Furthermore, if any d-precision arithmetic operation costs $F(d)$ units, then the worst-cost cost incurred is $xF(d)$. Thus, the worst-case runtime of bS is $O(F(d)log_2(d))$. □

4.2 Random Search (rS) for $n = 2$ Case

4.2.1 Description

In bS, the next value of $output_{22}$ to be parsed and checked is the point that bisects the search space. This occurs in Steps 5, 6 and 7, 8 of the bS algorithm. In rS, this value is sampled from a uniform distribution spanning a fixed search space, [0,1]. This is found in Step 4 of the following rS algorithm.

4.2.2 Algorithm

```
1. If $\pi_1 \geq \pi_2$ :
       $b_1 \coloneqq 0$
   Else:
       $1 - \frac{\pi_1}{\pi_2}$
2. $b_2 \coloneqq 1$ ; $iters \coloneqq 2000$ ; $output \coloneqq \begin{bmatrix} x_{11} & 1-p \\ 1-x_{11} & p \end{bmatrix}$
   where $p = \texttt{avg}([b_1, b_2]), x_{11} \sim U[0,1]$
3. While $output * \texttt{ev}(output) \neq \pi$ and $iters > 0$:
4.      $output_{22} = (b_2 - b_1) \cdot u + b_1$ where $u \sim U[0,1]$
5.      $output_{11} = 1 - (1 - output_{22})^{\frac{\pi_2}{\pi_1}}$
6.      $iters \mathrel{-}= 1$
7. Return $output$
```

4.2.3 Performance

The worst-case complexity for implementing rS is $O(F(d)10^d)$ with low probability, where $F(d)$ is the cost of calculating with arithmetic operations with d-digit precision.

Proof. Consider a complete Markov Chain of size $10^d \times 10^d$ with a transition matrix filled entirely of uniform distributions. That Markov Chain is analogous to iterating within rS, since each trial of u in Step 4 of rS is independent and uniformly random across all 10-digit combinations (digits 0-9) of d-digits. The probability of hitting the correct value is always fixed at $\frac{1}{10^d}$. We can call this probability p and use it as a parameter for a geometric distribution. The expected amount of steps until the correct value is reached is thus the expected value of this geometric distribution, which is $\frac{1}{p} = 10^d$. For each of these steps, an arithmetic cost of $F(d)$ is incurred, but with a variance of $\frac{1-p}{p^2} \gg 0$. Thus the worst-case complexity is $O(F(d)10^d)$ with low probability. $\square$

4.3 Binary-Random Search (brS) for $n = 2$ Case

4.3.1 Description

brS combines the efficiency of bS with the "open-mindedness" of rS. Instead of sampling uniformly from a fixed search space, as in rS, brS samples uniformly from the "remaining search space." The "remaining search space" is calculated by bisecting the previous "remaining search space," as in bS. Take note of Steps 4, 5 and 8, 9 to observe this in action:

4.3.2 Algorithm

```
1. If $\pi_1 \geq \pi_2$ :
        $b_1 \coloneqq 0$
   Else:
        $1 - \frac{\pi_1}{\pi_2}$
2. $b_2 \coloneqq 1$ ; $iters \coloneqq 2000$ ; $output \coloneqq \begin{bmatrix} x_{11} & 1-p \\ 1-x_{11} & p \end{bmatrix}$
   where $p = \mathtt{avg}([b_1, b_2]), x_{11} \sim U[0,1]$
3. While $output * \mathtt{ev}(output) \neq \pi$ and $iters > 0$:
4.          If $\mathtt{ev}(output)_2 < \pi_2$ :
5.                  $b_1 = \mathtt{avg}([b_1, b_2])$
6.                  $output_{22} \mathrel{+}= (b_1 - b_2) \cdot u$ where $u \sim U[0,1]$
7.             Else:
8.                  $b_2 = \mathtt{avg}([b_1, b_2])$
9.                  $output_{22} \mathrel{-}= (b_1 - b_2) \cdot u$ where $u \sim U[0,1]$
10.         $output_{11} = 1 - (1 - output_{22})\frac{\pi_2}{\pi_1}$
11.         $iters \mathrel{-}= 1$
12. Return $output$
```

4.3.3 Performance

The worst-case complexity for implementing brS is $O(F(d)log(d))$ with low probability, where $F(d)$ is the cost of calculating with arithmetic operations with d-digit precision. The proof follows the same reasoning as that of bS, because, on average, each remaining search space is expected to be bisected.

5 Finding $\mathbb{P}^{(I)}$ Directly when $n \geq 2$

In this section, we survey multiple means of directly calculating a $\mathbb{P}^{(I)}$ when $n \geq 2$, namely LP, NRS, GRS, GI, and CMA-ES.

5.1 Linear Programming (LP)

In this section, we describe how linear programs may be used to directly solve for $\mathbb{P}^{(I)}$.

5.1.1 Definition of LPs

A Linear Program (LP) is defined to be a vector $\vec{x}$ that minimizes or maximizes (the *objective*) some linear function $\vec{c}^T\vec{x}$ (the *objective function*) subject to *constraints* $\mathbf{M}\vec{x} \geq \vec{b}$ and $\vec{x} \geq \vec{0}$ where $\vec{x} \geq \vec{0} \iff \forall x \in \vec{x}, x \geq 0$. LPs are usually stated as follows:

$$\begin{aligned} &\textbf{[Objective] [Objective Function]} \\ &\text{s.t. } \mathbf{M}\vec{x} \geq \vec{b} \\ &\text{and } \vec{x} \geq 0 \end{aligned}$$

[WN]. For example:

$$\begin{aligned} &\textbf{Maximize } Z = x_1 + 2x_2 \\ &\text{s.t. } 5x_1 + 5x_2 \geq 20 \\ &\qquad\qquad x_2 \geq 4 \\ &\text{and } x_1, x_2 \geq 0 \end{aligned}$$

In this example,

$$c = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, M = \begin{bmatrix} 5 & 5 \\ 0 & 1 \end{bmatrix}, b = \begin{bmatrix} 20 \\ 4 \end{bmatrix}$$

And the solution occurs at $Z^* = 8, x_1^* = 0, x_2^* = 4$

Solutions to LPs are very easy to obtain using the *Simplex Algorithm*. Discussion of this algorithm is beyond the scope of this paper but is simple to implement.

5.1.2 Using LPs

We seek an n x n matrix $\mathbb{P}^{(I)}$ satisfying:

$$\mathbb{P}^{(I)}\pi = \pi \tag{3}$$

which is equivalent to finding all p_{ij} (the elements of $\mathbb{P}^{(I)}, \forall i, j = 1, 2, ..., n$) satisfying the following constraints:

$$\forall i = 1, 2, ..., n; \sum_{j=1}^{n} p_{ij}\pi_j = \pi_i$$

$$\forall j = 1, 2, ..., n; \sum_{i=1}^{n} p_{ij} = 1$$

$$\forall i, j = 1, 2, ..., n; p_{ij} \geq 0$$

Given:

$$\pi = [\pi_1, \pi_2, ..., \pi_n]^T$$

These constraints, in addition to the implied non-negativity constraints upon all p_{ij}, are almost enough to form an LP. All that is left to supply is an objective and linear objective function. For example:

Objective: Maximize

Objective Function: `trace`$(\mathbb{P}^{(I)})$

Intuitive Meaning: We want the most stable convergence to the optimal population distribution i.e. we want as few individuals moving as possible.

The diagonal elements of $\mathbb{P}^{(I)}$ represent the probabilities of individuals remaining in their bins between time steps. The larger the values of these elements, the higher the probability an individual does not move to another bin. Therefore, maximizing the trace of $\mathbb{P}^{(I)}$, the sum of these diagonal elements, is to maximize the probability that individuals in any bin do not move change their bin. Minimizing the movement of individuals is considered maximizing the *stability* of the population's convergence to the desired π.

As an additional consideration in this example, one should also add the constraint that any or at least some element of the diagonal of $\mathbb{P}^{(I)}$, p, should satisfy $p < 1$ to ensure that we do not output the identity matrix. The identity matrix not only maximizes the trace of $\mathbb{P}^{(I)}$ and is a permissible transition matrix (i.e. all columns are probability distributions), but it also satisfies:

$$\mathbb{P}^{(I)}\pi = \pi$$

In considering alternative pairs of objectives and objective functions, a common issue that arises is that the objective function must be linear, and this application has scant need for a linear objective function. Furthermore, the most useful candidate nonlinear objective functions may not lend themselves to an explicit form. For example:

Objective: Minimize

Objective Function: The second eigenvalue of $\mathbb{P}^{(I)}$

Intuitive Meaning: We want convergence to π to occur as fast as possible. The justification for this intuitive meaning is provided in Section 5.5.1.

We thus turn to methods that can either tolerate such difficult functions or do not rely on an objective function, but only after we discuss how to implement the above LP, which seeks to maximize `trace`$(\mathbb{P}^{(I)})$.

5.1.3 Implementing LPs

Appendix 10.1 provides Python code that may be used to solve for the above LP, which seeks to maximize `trace(`$\mathbb{P}^{(I)}$`)`. In this section, we discuss the method underlying the code.

We must first construct the linear objective function $\vec{c}^T\vec{x}$ where $\vec{x}$ is the vector of all elements in $\mathbb{P}^{(I)}$, otherwise known as the *vectorized* $\mathbb{P}^{(I)}$. Again, we seek to maximize `trace(`$\mathbb{P}^{(I)}$`)`, which is the sum of the diagonal elements of $\mathbb{P}^{(I)}$. Therefore, let:

$$\vec{c} = [c_1, c_2, \cdots, c_n]$$

where $\forall i \in \{1, \ldots, n\}, c_i[i] = 1, j \neq i \Rightarrow c_i[j] = 0$. So $\vec{c}$ looks like this:

$$\vec{c} = [1, 0, \cdots, 0, 1, 0, \cdots, 0, 0, 1, 0, \cdots, 0, 0, 0, 1, 0, \cdots\cdots, 0, 1]$$

Thus,

$$\vec{c}^T\vec{x} = \sum_{i=1}^{n} p_{ii}$$

To implement the first constraint, which requires $\forall j \in \{1, \ldots, n\}, \mathbb{P}^{(I)}[j, :]\pi = \pi[j]$, we let the matrix of linear constraints take the following form:

$$\mathbf{M} = \begin{bmatrix} \pi & \vec{0} & \vec{0} & \ldots & \vec{0} \\ \vec{0} & \pi & \vec{0} & \ldots & \vec{0} \\ \vec{0} & \vec{0} & \pi & \ldots & \vec{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vec{0} & \vec{0} & \vec{0} & \ldots & \pi \end{bmatrix}$$

where $\vec{0}$ is the n-length zero-vector.

We cannot forget to require that all columns of $\mathbb{P}^{(I)}$ sum to 1. Thus we must append a more constraints to the bottom of $\mathbf{M}$.

$$\mathbf{M} = \begin{bmatrix} \pi & \vec{0} & \vec{0} & \ldots & \vec{0} \\ \vec{0} & \pi & \vec{0} & \ldots & \vec{0} \\ \vec{0} & \vec{0} & \pi & \ldots & \vec{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vec{0} & \vec{0} & \vec{0} & \ldots & \pi \\ \vec{1} & \vec{0} & \vec{0} & \ldots & \vec{0} \\ \vec{0} & \vec{1} & \vec{0} & \ldots & \vec{0} \\ \vec{0} & \vec{0} & \vec{1} & \ldots & \vec{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vec{0} & \vec{0} & \vec{0} & \ldots & \vec{1} \end{bmatrix}$$

where $\vec{1}$ is an n-length vector of 1s. Thus,

$$\forall i = 1, 2, ..., n; \sum_{j=1}^{n} p_{ij}\pi_j = \pi_i, \;\; \forall j = 1, 2, ..., n; \sum_{i=1}^{n} p_{ij} = 1 \iff \mathbf{M}\vec{x} = \vec{b}$$

where

$$\vec{b} = \begin{bmatrix} \pi[1] \\ \vdots \\ \pi[n] \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

Note that $\mathbf{M}$ has dimensions $2n \times n^2$ and $\vec{b}$ has dimensions $2n \times 1$. Finally, we enforce the nonnegativity constraints:

$$\forall i, j = 1, 2, ..., n; p_{ij} \geq 0$$

If we choose to maximize $\texttt{trace}(\mathbb{P}^{(I)})$, then the identity matrix may be returned. To ensure we return an irreducible and nontrivial solution, we may enforce additional constraints on our parameters:

$$\forall i, j = 1, 2, ..., n; \epsilon < p_{ij} < 1 - \epsilon$$

for some small $\epsilon > 0$. Even with this, solutions from LP seem to resemble the identity matrix (all non-diagonal elements are ϵ, all diagonal elements are $1-\epsilon$). All solutions given by LP take this form, so although solutions are found nearly immediately in real-time, solutions are limited and often uninformative. Furthermore, we often find that the problem is infeasible likely due to floating point errors, even when we use methods designed to handle such errors like the "interior-point" method offered by the Python package, SciPy. Thus, unless we are more creative than the author in our choice of objective function, LP often fails us. Despite these shortcomings, LP may be useful for dealing with $\mathbb{P}^{(I)}$ with some constrained elements whenever it is feasible to implement such constraints.

We now turn to other methods that trade efficiency for always allowing us to find $\mathbb{P}^{(I)}$. For large n, these algorithms may perform especially slowly compared to LP, but they work nonetheless. Furthermore, iteration limits can be imposed to obtain approximately correct solutions and save runtime, which may qualify the following methods as more attractive options than LP.

5.2 Naïve Random Search (NRS)

5.2.1 Description

NRS can be viewed as an $n \geq 2$ generalization of rS. In each iteration of NRS, random columns of $\mathbb{P}^{(I)}$ are substituted for random column vector distributions from the n-dimensional uniform distribution.

5.2.2 Algorithm

Let $v =$ the current first eigenvector of $\mathbb{P}^{(I)}$.

```
1. Initialize all columns of $\mathbb{P}^{(I)}$ to be random samples from $U[0,1]^n$.
2. Until $\mathbb{P}^{(I)}$ has $\pi$ as its first eignenvector OR $iterations == max\_iterations$:
3.         For each column $p$ in $\mathbb{P}^{(I)}$:
4.                 Until $|p - p_{old}| < threshold$:
5.                         Sample $p_{new} \sim U[0,1]^n$
```

```
6.            If $\mathbb{P}^{(I)}$ with $p_{new}$ in place of $p$ has a first eigenvector closer to $pi$:
7.                $p \leftarrow p_{new}$
8. Return $\mathbb{P}^{(I)}$
```

5.2.3 Performance

Far more iterations are required to obtain the same accuracy as other methods. The variance of the worst-case runtime complexity is exponentially worse than rS courtesy of the Curse of Dimensionality, specifically $O(F(d)10^{dn^2})$. The only difference between this runtime and that of rS is the power of n^2, because now there are now n^2 matrix elements that must be randomly found whereas in rS there was only 1 value to find.

5.3 Greedy Random Search (GRS)

5.3.1 Description

GRS may be considered a generalized brS method. Let `normalize`(P) be the normalization function that ensures all columns of matrix P add to 1. Let `randInt`$(1:n)$ lazily return single iid samples from the discrete uniform distribution with support $\{1,\ldots,n\}$.

The intuition behind this algorithm is that if v_1 is the first eigenvector of $\mathbb{P}^{(I)}$ and if $v_1[a] < \pi[a]$, then we ought to ensure that the probability of staying at node a is greater than what it currently is. Likewise, if $v_1[a] > \pi[a]$, then we ought to ensure the probability of staying at node a is less than what it currently is. This logic was implemented, and, sadly, its effectiveness is limited. It rarely ever converges in a timely manner, hence why the number of iterations is limited. Often, the correct ordering of indices by size of their value in v_1 matches with that of those with values in π. For example, if $v_1 = [.35, .25, .4]$, then the indices are ordered, from least to greatest: 2, 1, 4. If $\pi = [.25, .05, .7]$, then the ordering of indices from least to greatest is still 2, 1, 4, and therefore the orderings of indices by their respective values in v_1 and π match. Again, for GRS, this form of accuracy is often the best it is typically capable of achieving, though there have been instances where it does work perfectly.

The most complex part of GRS occurs on Line 5 in the GRS Algorithm. In this line, we first ensure that if $\forall a \in \{1,\ldots,n\}; \mathbb{P}^{(I)}[a,a] < 0$ or $\mathbb{P}^{(I)}[a,a] > 1$, then we do not alter $\mathbb{P}^{(I)}$. Otherwise, we double-check that $\forall a \in \{1,\ldots,n\}; 0 < \mathbb{P}^{(I)}[a,a] < 1$, then alter $\mathbb{P}^{(I)}[a,a]$ by a factor of $(1+c)$. Finally, note that the value of 0.45 multiplied to c in Line 4 was selected purely because it seemed to work the best after some non-rigorous experimentation conducted by the author. Perhaps future research may find into the optimal value for this factor.

5.3.2 Algorithm

```
1. **Initialize** $\forall$ columns $\vec{p} \in \mathbb{P}^{(I)}, \vec{p} \sim U[0,1]^n$, normalize($\mathbb{P}^{(I)}$)
        $threshold = 0.001$
        $iterations = n * 100000$
2. **Until** $|\mathbb{P}^{(I)}\pi - \pi| < threshold$ **OR** $iterations == 0$ :
3.      $a \coloneqq$ randInt($1:n$)
4.      $c = 0.45 * (2 * (ev[a] < pi[a]) - 1)$
5.      $\mathbb{P}^{(I)}[a,a] = \mathbb{P}^{(I)}[a,a] * \Big(((1+c) * \mathbb{P}^{(I)}[a,a] > 1) + ((1+c) * \mathbb{P}^{(I)}[a,a] < threshold)\Big)$
            $+(1+c) * \mathbb{P}^{(I)}[a,a] * ((1+c) * \mathbb{P}^{(I)}[a,a] <= 1) * ((1+c) * \mathbb{P}^{(I)}[a,a] > 0)$
```

```
6.         normalize($\mathbb{P}^{(I)}$)
7.         $iterations$ $-= 1$
8. **Return** $\mathbb{P}^{(I)}$
```

Remember that logical operations may translate to integers in some programming languages, including Python e.g. $(1 < 2) \to 1, (1 > 2) \to 0$.

5.4 Gibbs-Inspired Method (GI)

5.4.1 Description

This algorithm derives its name from its inspiration, the Gibbs Sampler discussed in Geman & Geman [GG84]. Whereas the Gibbs Sampler persistently samples a randomly selected random variable from a distribution conditioned on all other random variables in consideration, GI persistently calculates the correctness of a matrix after optimizing a small, randomly selected part of that matrix.

5.4.2 Algorithm

```
1. **While** sum($|\mathbb{P}^{(I)}\pi - \pi|$) $<$ $tolerance$ AND $iterations < max\_iterations$:
2.         Isolate 2 random bins and the number of people within those 2 bins i.e. consider:

               $\mathbb{P}^{(I)}_{b_i,b_j}\pi_{b_i,b_j} = \pi_{b_i,b_j}$

           where

               $\mathbb{P}^{(I)}_{b_i,b_j}$ is $\mathbb{P}^{(I)}$ less all rows and columns but those with index $b_i$ or $b_j$.

           $\pi_{b_i,b_j}$ is the stationary distribution of $\mathbb{P}^{(I)}$ less all elements but those with index $b_i$ or $b_j$.

3.         Normalize $\pi_{b_i,b_j}$.
4.         Find the optimal $\mathbb{P}^{(I)}_{b_i,b_j}$ using bS, rS, or brS.
5.         Substitute these new values into the old values found at indices $b_i, b_j$, renormalizing the values with
respect to the amount of "mass" they cumulatively took up before within each of their respective columnar
probability distributions.
```

5.4.3 Performance

In practice, GI performs extremely well for small n and N. Relative to the other methods, GI is still fast for larger n and N and also exhibits a relatively good accuracy.

The Gibbs Sampling algorithm is considered to sample from the intended distribution when all parameters have been updated at least once (or a constant factor number of times). We will make the same assumption with GI. Given d-digit precision and when $n = 2$, bS will, at worst, cost some amount in $O(F(d)log_2(d))$. For $n > 2$, all possible pairs of row indices from $\mathbb{P}^{(I)}$ will eventually be subjected to bS. The expected time at which this occurs will be the time we calculate.

Define a Markov Chain $X = (X_1, X_2, ..., X_T)$ with $\binom{n}{2} + 1$ nodes with associated values $S = \{0, 1, ..., \binom{n}{2}\}$. These values represent the number of unique pairs of the indices $1, 2, ..., n$ parsed by GI. After setting $A := \binom{n}{2}$ and realizing that we uniformly choose among all possible pairs of indices, we can represent the Markov Chain graphically as follows:

$$\begin{tikzcd}
0 \arrow[r, bend right, "1"'] & 1 \arrow[loop above, "1-p_1"] \arrow[r, bend right, "p_1"'] & 2 \arrow[loop above, "1-p_2"] \arrow[r, bend right, "p_2"'] & \cdots \arrow[r, bend right, "p_{A-1}"'] & A \arrow[loop above, "1"]
\end{tikzcd}$$

where $\forall j \in \{0 \cdots A\}, p_j = \frac{A-j}{A}$. What we seek is $\mathbb{E}[T]$, where T is the total number of steps until GI completes. We embark on a *First-Step Analysis*, as it's called, to find $\mathbb{E}[T]$. If we set $v_j = \mathbb{E}[T|X_0 = j]$, and assert that we'll always be starting at $X_0 = 0$, then by the Law of Total Probability:

$$\mathbb{E}[T] = \sum_{j=0}^{A} v_j \mathbb{P}(X_0 = j) = v_0$$

because $\mathbb{P}(X_0 = 0) = 1$ and $\forall j = 1 : A, \mathbb{P}(X_0 = j) = 0$. Furthermore,

$$v_0 = 1 + v_1$$
$$\forall j \in \{1 \cdots (A-1)\}, v_j = 1 + (1-p_j)v_j + p_j v_{j+1}$$
$$v_A = 0$$

All that is unknown are the conditional expectations v_j. Note that to solve for them is equivalent to solving the system of equations $\mathbb{M}\vec{v} = \vec{v}$, where:

$$\mathbb{M} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & \dots & \dots & 0 \\ 1 & 0 & 1-p_1 & p_1 & 0 & \dots & \dots & 0 \\ 1 & 0 & 0 & 1-p_2 & p_2 & \dots & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & & 1-p_{A-1} & p_{A-1} \\ 0 & 0 & 0 & 0 & 0 & \dots & \dots & 0 \end{bmatrix}, \vec{v} = \begin{bmatrix} 1 \\ v_0 \\ v_1 \\ \vdots \\ v_A \end{bmatrix}$$

Note that the transition matrix for the Markov Chain X is embedded within $\mathbb{M}$ - if you remove the first column and last row of $\mathbb{M}$, you arrive at the transition matrix. In practice, it may be more computationally efficient to calculate the eigenvector of $\mathbb{M}$ corresponding to an eigenvalue of 1. Otherwise, we may continue to directly and analytically solve for v_0. We know from above that:

$$v_{A-1} = 1 + (1-p_{A-1})v_{A-1} + p_{A-1}v_A = 1 + (1-p_{A-1})v_{A-1}$$
$$\Rightarrow (1 + p_{A-1} - 1)v_{A-1} = 1$$
$$\Rightarrow v_{A-1} = \frac{1}{p_{A-1}}$$

Similarly,

$$v_{A-2} = 1 + (1-p_{A-2})v_{A-2} + p_{A-2}v_{A-1} = 1 + (1-p_{A-1})v_{A-1} + \frac{p_{A-2}}{p_{A-1}}$$
$$\Rightarrow (1 + p_{A-2} - 1)v_{A-2} = 1 + \frac{p_{A-2}}{p_{A-1}}$$
$$\Rightarrow v_{A-2} = \frac{1}{p_{A-1}} + \frac{1}{p_{A-2}}$$

.

$$\Rightarrow v_1 = \sum_{j=1}^{A-1} \frac{1}{p_j}$$

$$\Rightarrow v_0 = 1 + v_1 = 1 + \sum_{j=1}^{A-1} \frac{1}{p_j} = \sum_{j=0}^{A-1} \frac{A}{A-j}$$

$$\therefore \mathbb{E}[T] = A(\psi^{(0)}(A+1) + \gamma)$$

[Wol]

where $\psi^{(0)}$ is the 0^{th} derivative of the digamma function (i.e. the digamma function itself) and γ is the Euler-Mascheroni Constant. Again, the matrix form of the solution was provided as the reader may lack access to the digamma function or the Euler-Mascheroni Constant to a necessary precision.

Note that this expected runtime can easily be divided by s, where $s \in \{1 \cdots \frac{n}{2}\}$ is the number of computational nodes (e.g. servers, processors) available for parallelization of GI. Pairs of indices from $1, \cdot, n$ would be uniformly sampled without replacement. Each sampled pair would be assigned to a node, and bS (or rS or brS) would be computed on each node. The results from all nodes would be aggregated back into a central node, which would repeat the process of assigning random pairs of indices to nodes. One can also borrow from the field of Deep Learning and implement *dropout*, whereby each sampled pair of indices undergoes bS on a node only if $x = 1$, where $x \sim \text{Bernoulli}(p)$ and p is usually between 0.5 and 0.8. Droupout is conventionally used to prevent neural networks from overfitting their supplied training data. In this case, the author has no reason to suspect that performance enhancements, whether in accuracy or efficiency, may not be realized by implementing dropout.

5.5 CMA-ES Method

Covariate Matrix Adapation Evolution Strategy (CMA-ES) is a very general evolutionary algorithm that is meant to optimize "tricky" objective functions i.e. ones that are non-convex (lack a clear direction of descent toward any optima), multimodal (have multiple optima), non-smooth (derivatives of the objective function do not exist), discontinuous (the objective function and its derivatives do not exist), non-separable (there are dependencies between objective variables), noisy (the objective function does not exhibit any discernable pattern), or are high in dimension (the search space is too large, the problem is afflicted by the *Curse of Dimensionality*). These types of situations are known as *Black-Box Scenarios*, whereby the function cannot be accessed through traditional optimization approaches; All that's available to us may be some elements of the function's domain and range. Perhaps even the cost of evaluating the function eludes us. [Hanb] [Hanc] [Hana] [Hand]

We can define our objective function to be some function that accepts a matrix and outputs *true* if the matrix is a transition matrix (all elements are non-negative and all columns sum to 1) and has our desired eigenvalue of 1 with a corresponding eigenvector of π, else *false*. Perhaps we also seek to make the second largest eigenvalue in magnitude (the second eigenvalue) of the transition matrix as small as possible (see next subsection to learn why we may want this). Regardless of the objective function we choose, we are doomed to optimize a "tricky" function, and thus we must initialize a random $1 \times n^2$ vector, feed it to the CMA-ES, and hope that it computes fast enough. For floating-point precision less than 3 digits, the author can confirm that the CMA-ES does finish on a single processor in a reasonable amount of time.

5.5.1 A Word on Eigenvalues and Mixing Times

How do we know when an MC has reached its stationary distribution? There may exist plenty of fluctuations in the behavior of $(\mathbb{P}^{(I)})^t \vec{x}$ for some initial distribution $\vec{x}$ and increasing $t > t_0 > 0$, even if we already reached

$(\mathbb{P}^{(I)})^{t_0}\vec{x} = \pi$ for some t_0. The time at which the distance between the current distribution $(\mathbb{P}^{(I)})^t\vec{x}$ and the stationary distribution π is ϵ-small is called the *mixing time* and is defined as follows:

Definition 9 (Mixing Time)**.** *Let $\vec{x}$ be any positive semi-definite vector, $\forall i \in \{1 \cdots n\}, \vec{x}[i] \geq 0$, satisfying $\sum_{i=1}^{n} \vec{x}[i] = 1$. Choose an $\epsilon > 0$. Then the mixing time is:*

$$t_{mix}(\epsilon) = \min\{t : d(t) \leq \epsilon\}$$

where

$$d(t) = \|(\mathbb{P}^{(I)})^t\vec{x} - \pi\|$$

With this concept in mind, we can develop means of at least limiting the magnitude of $t_{mix}(\epsilon)$ beyond relaxing ϵ. Define:

$$\pi_{min} = \min_{i\in\{1\cdots n\}} \pi[i]$$

and let λ_2 be the second largest eigenvalue of $\mathbb{P}^{(I)}$. It can be shown that:

Theorem 2.

$$t_{mix}(\epsilon) < 1 + \frac{1}{1-\lambda_2}\log\left(\frac{1}{\epsilon\pi_{min}}\right)$$

[Kal]

After noting that all eigenvalues of a transition matrix are elements of [0,1], then, for a given ϵ, we know that minimizing λ_2 (getting it as close to 0 as possible) will minimize $t_{mix}(\epsilon)$, and, inversely, maximizing λ_2 (getting it as close to 1 as possible) will maximize $t_{mix}(\epsilon)$.

Additionally, it can be shown that the greater the *eigengap* $|\lambda_1| - |\lambda_2|$ the more stable the stationary distribution is to perturbations in the MC. [HK03]

Perhaps we want our population to converge to π as quickly as possible, and perhaps we want its convergence to π to be as stable as possible. We can then use both previously stated facts regarding the eigenvalues of $\mathbb{P}^{(I)}$ as *regularizations* in our objective function. In other words, they can be used to weight the output of our objective function in favor of certain properties. Again, the exact mapping from $\mathbb{P}^{(I)}$ to its eigenvalues exhibits a high-dimension argument space (its arguments are every element of $\mathbb{P}^{(I)}$), is non-separable (the elements of $\mathbb{P}^{(I)}$ interact ambiguously to generate the eigenvalues), and, for these and further reasons, may simply be classified as "tricky". Thus, the call to care for these regularizations becomes a call to utilize the best means available to deal with them, the CMA-ES algorithm.

See the code in Appendix 10.8 for an implementation of CMA-ES in Python. The Python module `cma`, which includes all necessary CMA-ES functionality, is maintained by Nikolaus Hansen of the Université Paris-Saclay. [Hanb] [Hanc] [Hana] [Hand]

5.6 Direct Methods with Constrained Parameters

All methods allow us to find optimal $\mathbb{P}^{(I)}$ with constrained parameters, where some of the elements of $\mathbb{P}^{(I)}$ hold already-specified values and cannot be altered. We assume that the addition of constraints is intuitive for the "n=2"-case methods, LP, CMA-ES and NRS. Constraints are deeply intertwined and go hand-in-hand LP and CMA-ES. NRS can be simply modified to incorporate additional constraints; For each column of $\mathbb{P}^{(I)}$ with k constrained elements, just reject all normalized samples from $Uniform[0,1]^n$ whose values at the k indices are not permissible. GRS and GI may lack this intuitiveness in the face of constrained parameters.

5.6.1 Constrained GRS Example

Given a dictionary that lets us check if a column has constraints, the following procedure would be inserted between Line 5 and Line 6 of the GRS algorithm. For each element with a constraint i in the selected column k, randomly sample a value within an interval $[a_i, b_i]$ satisfying $0 \leq a_i \leq b_i \leq 1$. Call this sample c_i. Note that when $a_i = b_i$, we have an equality constraint. Otherwise, we have an inequality constraint. Normalize the column by multiplying all non-constrained values in column k by $1 - \sum_i c_i$. Proceed as usual.

5.6.2 Constrained GI Example

Within GI, we persistently form various $n = 2$ subproblems. We check various permissible values for p_{22}, solve for p_{11} using p_{22}, return the result to the original problem and repeat. The space of possible values for p_{22} is already constrained to ensure that $p_{11} + (1 - p_{11}) = p_{22} + (1 - p_{22}) = 1$ and $p_{11}, p_{22} > 0$. We can simply further constrain this space to incorporate our additional constraints. In the case of equality constraints, Equation (1) subject to Constraint (2) can be used to immediately recover p_{11} whenever p_{22} is given and vice versa. In the case of inequality constraints, we can still use Equation (1) but subject it to Constraint (2) in addition to whatever other constraints we must implement, and take pause when all constraints cannot be satisfied at once. Here is an example when all constraints can be simultaneously satisfied:

Suppose we must enforce that $p_{13} < \frac{1}{2}$, then whenever p_{13} is selected to be p_{11} in one of many $n = 2$ subproblems of a GI instance, we randomly select a value p_{22} such that:

$$p_{22} \geq 1 - \frac{\pi_1}{\pi_2}$$

and

$$p_{11} = 1 - (1 - p_{22})\frac{\pi_2}{\pi_1} < \frac{1}{2}$$

We can manipulate that second, additional constraint in the following manner:

$$\frac{1}{2} < (1 - p_{22})\frac{\pi_2}{\pi_1}$$

$$\frac{\pi_1}{2\pi_2} < (1 - p_{22})$$

$$p_{22} < \frac{\pi_1}{2\pi_2}$$

Therefore, we randomly choose a p_{22} such that:

$$1 - \frac{\pi_1}{\pi_2} \leq p_{22} < \frac{\pi_1}{2\pi_2}$$

ensuring always that:

$$0 \leq p_{22} \leq 1$$

Here and in all cases, the values in π – in this case, π_1 and π_2 – determine whether or not such a p_{22} value even exists (whether or not the constraints admit a *feasible* solution).

6 Relaxing the Assumptions of Direct Methods

6.1 Relaxing the Identityless Assumption

If individuals are expected to contribute to nodal functions or the welfare function to varying degrees, then we ought to drop the Identityless Assumption. If the number of types of individuals is C, then we will have C equations:

$$\forall i = 1, ..., C; \mathbb{P}_i^{(I)} \pi_i = \pi_i \tag{4}$$

where $\mathbb{P}_i^{(I)}, \pi_i$ refer to the transition matrix and stationary distribution for the ith "type" of unique individual, and are $n \times n$ and $n \times 1$, respectively. Each of these C equations can be solved independently of the others.

6.2 Relaxing the Independence Assumption

The simplest form of dependency can be introduced without much effort. Consider a situation where both Identityless and Independence Assumptions are relaxed. You can say that certain types of individuals get to "choose their distribution about the bins first" and the optimal stationary distribution for other types of people is calculated thereafter. This task of finding the stationary distributions for each type of individual "in order" does not concern this paper as it would lead to a situation no different than that described by Equations (4). But this situation is not truly dropping the Independence Assumption. Although dependencies encoded in the calculation of stationary distributions has been introduced, dependencies influencing the transitions of individuals as they are transition between bins are more pragmatic and have yet to be tackled.

With the Independence Assumption actually dropped, all of a sudden the importance of a single individual's movements is lost – $\mathbb{P}_{ij}^{(I)}$ and π_i become useless. We now need information pertaining to the positions of all individuals, and how each of their positions affects the positions of other individuals. This motivates the development of the *Mass Matrix*, $\mathbb{P}^{(M)}$, which allows us to insert information regarding dependencies between individuals or types of individuals. In this paper, when we drop the Independence Assumption, we will primarily be concerned with dynamics that arise from a population identityless individuals; We will tend to not drop Independence Assumption unless we maintain the Identityless Assumption.

7 Finding the Mass Matrix ($\mathbb{P}^{(M)}$)

In this section, we define the *Mass Matrix*, $\mathbb{P}^{(M)}$, motivate its existence, and provide two algorithms one can use to generate a $\mathbb{P}^{(M)}$, namely VSEA and Series Method. With each algorithm comes a description of how it works along with a quick word on its performance. Within the Series Method, we also define the notion of *successively-bound weak compositions* and ways in which we may use $\mathbb{P}^{(M)}$ and the Series Method in relaxing the Independence Assumption.

7.1 Defining $\mathbb{P}^{(M)}$

In this section, we formally define $\mathbb{P}^{(M)}$ and provide an intuitive meaning of its structure.

$\mathbb{P}^{(I)}$ is an $n \times n$ transition matrix that describes how each individual hops about bins. On the contrary, $\mathbb{P}^{(M)}$ describes how the entire population hops about different distributions and is an $s \times s$ transition matrix where:

$$s := \binom{N + n - 1}{n - 1}$$

So we may say that $\mathbb{P}^{(M)}$ is a function mapping $(n, N) \mapsto M$, where $M \in \mathbb{M}^{s\times s}$, the space of $s \times s$ matrices. Consider the following generalized $\mathbb{P}^{(M)}$:

$$\begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1s} \\ q_{21} & q_{22} & \cdots & q_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ q_{s1} & q_{s2} & \cdots & q_{ss} \end{bmatrix}$$

What do each of these q_{ij} values mean? Each is the probability of the entire population changing from distribution j to distribution i in a single time step.

The index corresponding to each distribution is arbitrary but must be consistent. For instance, if $n = 2$ and $N = 3$, index 1 may correspond to the population distribution where 3 people in bin 1 and no people are in bin 2 (let's denote this state as [3,0]). Index 3 may correspond to the population distribution [1,2]. Thus, the element in column 1, row 3 of $\mathbb{P}^{(M)}$ will be the probability of the 3 people hopping from state [3,0] to [1,2].

It ought to be noted that the weighted sum $\pi = \sum_{j=1}^{s} v_j \vec{d_j}$, where v_j is the jth element of the first eigenvector of $\mathbb{P}^{(M)}$, v, and $\vec{d_j}$ is the possible population distribution associated with index j of $\mathbb{P}^{(M)}$. It is this property that powers the reasons we are motivated to generate $\mathbb{P}^{(M)}$.

7.2 Why do we care about $\mathbb{P}^{(M)}$?

In this section, we discuss why we may want to find $\mathbb{P}^{(M)}$.

First, we may use $\mathbb{P}^{(M)}$ to verify our results (the values of $\mathbb{P}^{(I)}$ we obtain from any of the above algorithms). Within the first eigenvector (corresponding to the largest eigenvalue, 1) of $\mathbb{P}^{(M)}$, some element will correspond to π, the ideal stationary distribution of $\mathbb{P}^{(I)}$ we seek to impose. If we maximize that element, then we maximize the probability of achieving that stationary distribution. If our model lacks a population size sufficient to arrive at π, then we expect that the weighted sum of all possible population distributions, weighted by their corresponding entries of the first eigenvector of $\mathbb{P}^{(M)}$, will approximate π. Furthermore, we suspect that the possible population distributions "closest" to π (e.g. by *cosine distance*) ought to have the largest corresponding entry values in that first eigenvector.

Secondly, $\mathbb{P}^{(M)}$ gives us an opportunity to generate new, perhaps more powerful solutions since it allows us to invoke new regularizations. This feature of $\mathbb{P}^{(M)}$ only applies when we need an objective function. For example, we may choose our objective function in the CMA-ES algorithm to weight the minimization of the second eigenvalue of $\mathbb{P}^{(M)}$ or, additionally, to weight the maximization of the element of the first eigenvector of $\mathbb{P}^{(M)}$ corresponding to π.[1]

Finally, the generation of $\mathbb{P}^{(M)}$ provides a stepping stone one may use toward relaxing the Independence Assumption. We will discuss this in Section 7.4.5.

7.3 Vector-Sum Enumeration Algorithm (VSEA)

In this section, we attempt to provide sufficient clarity into the algorithm coded in Appendix 10.9. Consider the following subsection to be a line-by-line analysis of the coded algorithm.

The VSEA was the author's first attempt to tackle the problem of generating $\mathbb{P}^{(M)}$. It is currently not the preferred means of generating $\mathbb{P}^{(M)}$, but it is included anyway to demonstrate the author's development

[1]For the purposes of efficiency, it should be noted that symbolic math packages exist (such as SymPy `http://www.sympy.org/en/index.html`) that would greatly facilitate the extraction of information from $\mathbb{P}^{(M)}$. If, for instance, the second eigenvalue of $\mathbb{P}^{(M)}$ is included in the objective function of the CMA-ES algorithm, one only needs to create the symbolic $\mathbb{P}^{(M)}$ once and take aMantage of the fact that $\mathbb{P}^{(M)}$ depends entirely on $\mathbb{P}^{(I)}$. Once new values for $\mathbb{P}^{(I)}$ have been obtained in successive iterations of the CMA-ES algorithm, the updated $\mathbb{P}^{(M)}$ can be quickly generated from the symbolic $\mathbb{P}^{(M)}$.

and to provide others with a starting point from which they may find a means of calculating $\mathbb{P}^{(M)}$ with an efficiency overshadowing that of the VSEA and perhaps even the currently preferred means of generating $\mathbb{P}^{(M)}$ (to be discussed later).

In the following description, "Block X" refers the reader to the section in the code in Appendix 10.9 labelled with the in-line comment "Block X".

7.3.1 Description of VSEA

In this section, an explanation of how VSEA operates is provided.

Block A

Initialize some $s \times s$ matrix of zeros. This matrix will eventually be filled-in with the correct values and outputted. Create a "basis" (we don't mean "basis" in the typical sense) of all vectors that each represent a shift between possible mass states and describe the exact amounts of people lost and gained in each bin between transitions. For example, the basis vector representing moving someone from bin 1 to bin 2 when $n = 3$ is $[-1\ \ 1\ \ 0]$. A zero vector ought to be initialized as well. These are not to be added to the initialized basis, but used later, whenever a person may stay within a bin. For instance, if our system moves from mass state $[2\ \ 0]$ (2 people in bin 1 and 0 people in bin 2) to $[1\ \ 1]$, then we ought to later use one zero vector, $[0\ \ 0]$ representing the lack of movement of one individual in bin 1.

Block B

Iterate over all possible population-state transitions i.e. over all elements in some zero-initialized $\mathbb{P}^{(M)}$. At each step, we will find the probability of that transition within one time step. For example, when $n = 2, N = 7$, what is the probability of going from a state with 4 people in bin 1 and 3 people in bin 2 to a state with everyone in bin 2?

Each state is a weak composition $\texttt{WC}(N, n)$, so $\mathbb{P}^{(M)}$ can also be considered the collection of probabilities describing movements between any two weak compositions generated with parameters N, n. From this point forward, we will refer to the current state as T_j and the new state as T_i.

Block C

Determine the element-wise difference between 2 states. For each bin $k \in \{1 \cdots n\}$, calculate the number of zero-vectors required to model the state transition, which equals $\min(T_i[k], T_j[k])$. Then, associate each 0 vectors with the corresponding diagonal element from $\mathbb{P}^{(I)}$. For instance, if anyone may stay in bin 2 between states, then we associate $p^{(I)}_{22}$ with a zero-vector.

Block D

Associate each basis vector with an element from $\mathbb{P}^{(I)}$. Given a population of N individuals, we may imagine that there are N tokens to be distributed, with replacement, among all basis vectors, which now include the added zero-vectors. Each distribution of tokens needs to be considered, so iterate over all possible distributions. It should be noted that each token distribution is another set of weak compositions with parameters $N, lvlc$, where $lvlc$ is the total number of basis vectors for element $\mathbb{P}^{(M)}_{i,j}$. Each of theses weak compositions of tokens may be a possible means of achieving the state transition $T_j \to T_i$. If we take the weighted vector sum of the basis vectors, with each vector weighted by their respective number of associated tokens, then we ought to arrive at the vector difference $T_j - T_i$, hence the name "Vector-Sum Enumeration Algorithm." In the next algorithm, we will see that the major runtime improvement comes from limiting this set of weak compositions to only the necessary weak combinations that actually achieve the relevant state transition.

If a token distribution places too many tokens in the "bins" associated with any of the zero-vectors, then the token distribution is not permissible, and we move on to the next token distribution. We can check for permissibility by taking a weighted vector sum of the basis, with the weights for each vector being the number of tokens assigned to that vector. If that weighted sum equals the difference between mass states (e.g. If you are currently considering the state transition between State_1 = [2 0] and State_2 = [1 1], then the difference is $[-1\ \ 1]$ = State_2 - State_1) then keep that weighted sum.

Block E

If a distribution is permissible and there are not too many people leaving any bin after the basis vectors have been applied (this needs to be explicitly checked), then we begin to keep track of who goes where i.e. we begin to answer: How many people are actually entering each bin, and when people leave each bin, where do they end up? This will be used in the next block when we determine the number of ways this particular transition $T_j \to T_i$ can occur. For instance, if we transition from [13] $\to$ [22], Block D will begin to iterate through the following situations:

- 1 person stays in bin 1, 1 person moves from bin 2 to bin 1
- 1 person moves from bin 1 to bin 2, 2 people move from bin 2 to bin 1

whereas Block E create the necessary data structures to determine that:

- There $\binom{3}{1} = 3$ ways that 1 person stays in bin 1 (with probability p_{11}) and 1 person moves from bin 2 to bin 1 (with probability p_{21}) (1 of any 3 different people can be chosen to move from bin 2 to bin 1)
- There are $\binom{3}{2} = 3$ ways that 1 person moves from bin 1 to bin 2 (with probability p_{21}), 2 people move from bin 2 to bin 1 (with probability p_{21}^2) (2 of any 3 different people can be chosen to move from bin 2 to bin 1)

Finally, Block E maps basis vectors to their corresponding element in $\mathbb{P}^{(I)}$. The final calculation for this mass matrix element will be a sum, where each term consists of an integer coefficient, elements from $\mathbb{P}^{(I)}$, and exponents. The coefficients will be determined in Block F, Block E maps basis vectors to elements in $\mathbb{P}^{(I)}$, and the exponents acting on these elements of $\mathbb{P}^{(I)}$ are the number of tokens allotted to each basis vector in Block D. In continuing our example where $[1\ \ 3] \to [2\ \ 2]$:

$$\mathbb{P}^{(M)}_{ij} = 3p_{11}p_{21} + 3p_{12}p_{21}^2$$

where $p_{11}, p_{12}p_{21}, p_{12}$ are all elements of $\mathbb{P}^{(I)}$.

Block F

Substitute each basis vector for its associated value in $\mathbb{P}^{(I)}$ and let that value be raised to a power equal to the number of tokens allotted to the value's respective basis vector. Count the number of ways in which a particular migration of individuals can occur using the data structures formed in Block E. This number will become our coefficient for the current set of selected basis vectors. For each bin i, our coefficient is $\binom{k^{(i)}}{k_1^{(i)}, k_2^{(i)}, \ldots}$, where $k^{(i)}$ is the total number of individuals within a bin i and $k_1^{(i)}$ is the total number of those individuals leaving bin i and entering bin 1. This is a multinomial choose operation that will become extremely important in the next algorithm.

We repeat from Block D for all ways of attaining $T_j \to T_i$ and we repeat from Block B for all s^2 elements in $\mathbb{P}^{(M)}$.

7.3.2 Performance

This algorithm is $\in O(s^2tl)$, since the algorithm requires looping through all $s^2 = \binom{N+n-1}{n-1}^2$ elements of $\mathbb{P}^{(M)}$, all $t = \binom{N+l-1}{l-1}$ potential basis vectors per element of $\mathbb{P}^{(M)}$, and all $l = (\frac{n!}{(n-n)!} + N) = n! + N$ basis vectors (each set of basis vectors has size l and is a list of all permutations of -1, 1, and $n-2$ zeros plus, at most, 1 zero vector for every bin 1. . . n). To fully grasp the disgusting magnitude of this runtime, we may write it as:

$$O\left(\binom{N+n-1}{n-1}^2\binom{2N+n!-1}{N+n!-1}(n!+N)\right)$$

7.4 Series Method

As promised, we improve on the VSEA by limiting the number of permissible weak vector combinations of basis vectors. This task is difficult because one must know all the possible ways each individual from any bin may be allotted among all bins in the next time step given the movements of individuals from all previously parsed bins. The main insight that allows for the Series Method to exist is observing that the many ways individuals may be allotted to new bins is equivalent to some subset $W_i \subset \texttt{WC}(b[i], n)$. The permissible weak compositions from bin i consist of elements of $\{w \in W_i | w + \sum_{j=1}^{i} k_j \leq T_i, \forall j \in \{1 \dots n\}, k_j = W_j\}$ i.e. we only choose weak compositions from W_i that do not cause any bins to "overflow" with more individuals than the upper limit prescribed by T_i. Luckily, we have created an algorithm that can generate all possible permissible weak compositions, and we call the results of this algorithm a set of *successively-bounded weak compositions.*

7.4.1 Successively-Bounded Weak Compositions

In this section we define the notion of successively-bounded weak compositions.

Definition 10 (Successively-Bounded Weak Composition)**.** *A successively-bounded weak composition, $L \in$ $\texttt{SBWC}(v, w)$, with support vector v and resistance vector w satisfying $\sum_{i=1}^{n} w_i \geq \sum_{i=1}^{n} v_i$, is a list L of n, n-length vectors satisfying:*

$$\forall i = 1, ..., n; \sum_{j=1}^{n} \vec{L}_i[j] = v_i$$

and

$$\sum_{i=1}^{n} L_i \leq w$$

where n is the length of both v and w. $\vec{L}_i[j]$ represents element j of vector $\vec{L}_i$. Equality in the second constraint occurs when $\sum_{i=1}^{n} w_i = \sum_{i=1}^{n} v_i$.

Note that the first summation denotes scalar addition whereas the second denotes vector addition.

7.4.2 Description

In this section, an explanation of how the Series Method operates is provided. The Series Method is an implementation of the following series:

$$\mathbb{P}^{(M)}_{s^{(2)}, s^{(1)}} = \sum_{i \in \texttt{SBWC}(s^{(1)}, s^{(2)})} \left[\prod_{k=1}^{n} \binom{s_k^{(1)}}{i}\left[\prod_{t=1}^{n} p_{t,k}^{i_t}\right]\right] \tag{5}$$

where $\texttt{SBWC}$ and $\texttt{WC}$ are defined above and $p_{t,k}$ are elements of $\mathbb{P}^{(I)}$. It should be noted that every i is a vector and that $\binom{s_k^{(1)}}{i}$ is a multinomoial choose operation. The intuition empowering this series formula is hinted within the description of the VSEA, but is stated explicitly below.

7.4.3 Intuition

In this section, an intuitive explanation of why the Series Method operates successfully is provided.

In transitioning from $s^{(1)}$ to $s^{(2)}$, every bin in $s^{(1)}$ may distribute its contents among any of the bins in any permissible way. The one constraint restricting the number of these possible distributions is the *resistance*, $s^{(2)}$ and the cumulative allotment of individuals donated by previously parsed bins. In other words, the total amount of "content" gifted to any particular bin from all bins in $s^{(1)}$, the *support*, must equal the amount of "content" that bin ought to contain in $s^{(2)}$.

The total number of ways of using a particular set of individuals' movements to achieve any particular distribution of individuals $s^{(2)}$ from $s^{(1)}$ is given by $\prod_{k=1}^{n} \binom{s_k^{(1)}}{i}$ for a given successively-constrained weak composition i. The probability of any single one of these sets of movements occurring is given by $\prod_{t=1}^{n} p_{t,k}^{i_t}$, where all $p_{t,k}$ are elements of $\mathbb{P}^{(I)}$.

7.4.4 Performance

The Series Method has a runtime of $O(Q(N,n) \cdot G(N,n) \cdot n^2 \cdot \binom{N+n-1}{n-1}^2)$. This can be derived by simply looking at Equation (5) and noting that the summation and two product operators are embedded within one another. To get one out of the many $\binom{N+n-1}{n-1}^2$ elements of $\mathbb{P}^{(M)}$, n calculations in the innermost product operators must be calculated n times over, by virtue of the second product operator, which must in turn be computed $Q(N,n)$ times over by virtue of the summation. $Q(N,n)$ is the total number of *successively-bounded weak compositions supported* by v and *resisted* by w where $(v,w) \in \texttt{WC}(N,n) \times \texttt{WC}(N,n)$. In particular:

$$Q(N,n) := |\{\texttt{SBWC}(v,w)\,|(v,w) \in \texttt{WC}(N,n) \times \texttt{WC}(N,n)\}|$$

Finally, $G(N,n)$ is the worst-case complexity in generating $\texttt{SBWC}(s^{(1)}, s^{(2)})$ [2]. In lieu of providing a detailed worst-case runtime complexity for the Series Method, its real-time runtime for various inputs is recorded in a later section alongside all other methods.

7.4.5 Relation to Relaxation of the Independence Assumption

Two ways of dealing with the relaxation of the Independence Assumption are outlined below:

1. Quick-and-Dirty Approach: We may choose to remove certain elements from SBWC, suggesting that only certain distributions of individuals are allowed and the calculated $\mathbb{P}^{(M)}$ can be normalized after the series calculation has touched every element in $\mathbb{P}^{(M)}$ (the "Quick" part). There's little theory backing the efficacy of this process and it does not generalize to situations involving a relaxed Identityless Assumption (the "Dirty" part).

2. Formal Approach: This process allows for the relaxation of both core assumptions of the direct methods but is bound to live just as theory for the foreseeable future as it would require a massive amount of data to put into practice. Consider again Equations (4), only this time, let $\mathbb{P}_i^{(I)}$ and π_i be massively larger than before. They now become very-high-dimensional tensors. Let's consider a supposedly simple case, when $n = 2$. Each element of $\mathbb{P}_1^{(I)}$ would be the probability of individual of type 1 moves from one bin to another given the presence of a certain number of type 2 individuals in bin 1, a certain number of type 2 individuals in bin 2, certain number of type 3 individuals in bin 1, etc. Clearly, the relaxation of this assumption quickly becomes unbearable despite being theoretically possible.

[2] The algorithm generating $\texttt{SBWC}(s^{(1)}, s^{(2)})$ may be sped up from the insights provided by Page's work on restricted weak compositions: [Pag12]

8 Summary of Methods and Performance

Here, we list charts associating methods to their worst-case or expected runtimes.

For all subsections within this section:

- L is maximum bit-length of any number in input, with randomized Simplex Algorithm.
- d is the employed digit precision.
- $\psi^{(0)}$ is the 0^{th} derivative of the digamma function (i.e. the digamma function itself).
- γ is the Euler-Mascheroni Constant.

We also provide the real-time runtimes of each method that result from using various parameter or input values so we may better understand how each method performs in practice.

8.1 Methods for Finding an Optimal $\mathbb{P}^{(I)}$ when $n = 2$

In this section, we list the runtime complexities (the worst-case runtime for deterministic methods and expected runtime for probabilistic methods) for all "$n = 2$"-case direct methods. We then show their real-time performances for various parameter values.

8.1.1 Complexities

In practice, rS, bS, and brS all finish execution in a matter of seconds. All direct methods were tested on a computer with a single processor with values of $n \in 2, \ldots, 14$.

Method	Complexity
rS	$O(F(d)10^d)$
bS	$O(F(d)\log(d))$
brS	$O(F(d)\log(d))$

8.1.2 Real-Time Runtimes

All direct methods have a threshold parameter ϵ that determines whether or not our solutions is sufficiently correct. In particular, we run each algorithm until

$$\|P\pi - \pi\| < \epsilon$$

where P is the candidate value for $\mathbb{P}^{(I)}$ for a given π. In this section, we vary that ϵ parameter and note the differences in real-time runtimes that consequently surface between "$n = 2$"-case direct methods. Figure (1) plots the real-time runtimes for bS, rS, and brS for various epsilon values. For each epsilon, we average the real-time runtimes of each algorithm across 300 trials on the same "$n = 2 \times 1$"-size π. In Figure (2), we plot the same information with 100,000 trials. The anticipated indirect relationship is far more apparent in Figure (2). In both cases, rS performs relatively worse than bS and brS though not by much, and brS performs slightly better than bS.

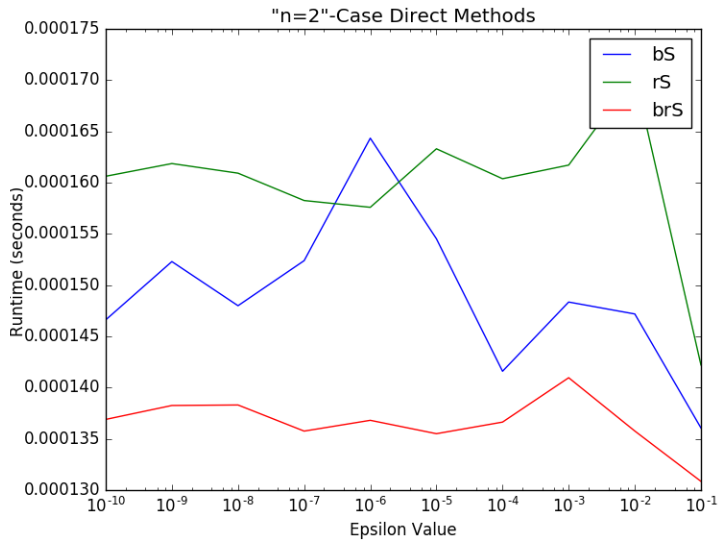

Figure 1: With 300 trials

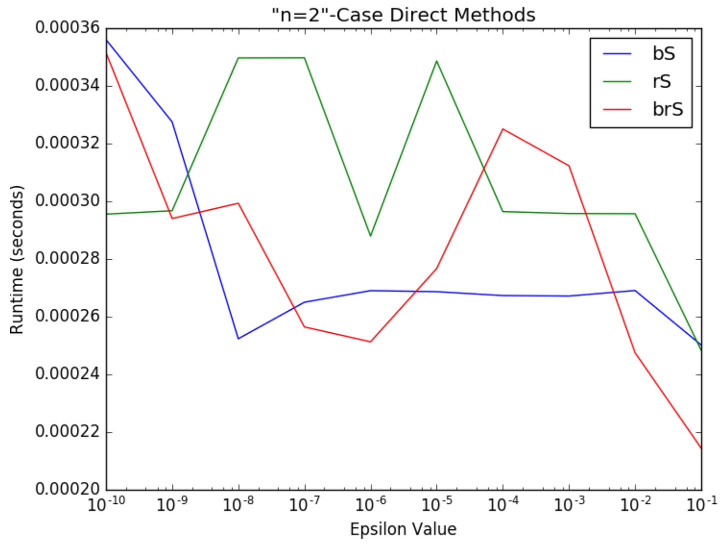

Figure 2: With 100,000 trials

8.2 Methods for Finding an Optimal $\mathbb{P}^{(I)}$ when $n \geq 2$

In this section, we list the runtime complexities (the worst-case runtime for deterministic methods and expected runtime for probabilistic methods) for all $n \geq 2$ direct methods. We then show their real-time performances for various inputs. We divide this section into 2 subsections: reliable (LP, GI) and unreliable methods (NRS, GRS, CMA-ES). The former class tend to always finish execution for small n on a single processor in a reasonable amount of time. The latter class tends to be less dependable in this respect. Remember that for all methods in all classes except LP, iteration limits can be imposed to sacrifice accuracy for a lower runtime.

8.2.1 Complexities

Here, we list the worst-case or expected runtimes for all $n >= 2$-case direct methods.

Method	Complexity
LP	$O(n^3 L)$ with high probability [LBE11] [ST08] [KS]
NRS	$O(F(d)10^{dn})$
GRS	N/A
GI [Wol]	$\binom{n}{2}(\psi^{(0)}(\binom{n}{2}+1)+\gamma)$
CMA-ES	N/A

8.2.2 Real-Time Runtimes

Here, we provides graphs showing the real-time runtimes for all $n >= 2$-case direct methods for various n values. In the case of the reliable methods, for each $n \in \{2, 3, 4, 5\}$, we plot the average runtime over 8 trials vs. the n-value. In the case of the unreliable methods, for each $n \in \{2, \ldots, 10\}$, we plot the proportion of times the algorithm finishes running in $60n$ seconds.

Reliable Methods (LP, GI):

LP, with the objective of maximizing the objective function $\texttt{trace}(\mathbb{P}^{(I)})$, was by far the fastest algorithm but only returned results resembling the Identity Matrix. The diagonal elements were often $1 - t(n-1)$, where t is the tolerance used to measure floating-point equivalence (floating-point numbers a, b satisfy $a = b \iff |a - b| < t$) and the non-diagonal elements tended to be t. This rule held true with minimal exceptions.

GI, on the other hand, did not always converge. The author noticed that sometimes, for the same n, GI would converge in less than a second for multiple trials. Occasionally, and especially as n increased, GI would appear to get "stuck" and either take a very long time to converge or not ever converge. The author noted that the variance in seconds of real-time runtimes for repeated GI trials where $n = 5$ was an astounding 13151.8180598. The author speculates that parallelzing GI, implementing dropout, or simply restarting GI if it exceeds a certain time limit ought to all help mitigate this issue.

The expected runtime of GI appeared to be predictive of the actual runtime of GI. Figures (5) and (6) show GI vs. the Expected Runtime of GI (labelled as "Expected GI"), and (7) notes the ratio between the actual runtime GI over the Expected GI. The author suspects that this ratio is a rugged, generally increasing function because of the presence of "outlier trials" where GI would seem to get stuck. It is these same "outlier trials" that account for the large variance mentioned earlier.

Unreliable Methods (NRS, GRS, CMA-ES):

NRS, GRS, CMA-ES were all considered unreliable because they generally do not produce an output as quickly as LP or GI. Here, we measured the proportion of times out of 8 trials each method returned a solution in $60n$ seconds per each $n \in \{2, \ldots, 5\}$. NRS performed miserably, never once returning a solution in the allotted time. CMA-ES performed well for $n < 4$, though it did not return a single value in the allotted time once $n > 3$. GRS performed the best out of all of these functions, returning a solution for all n except when $n = 3$, inexplicably. Figure (8) plots these outcomes.

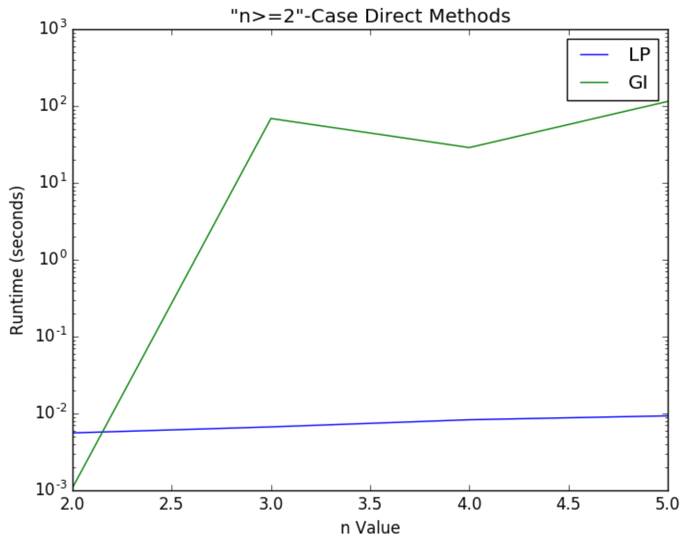

Figure 3: LP and GI real-time runtimes

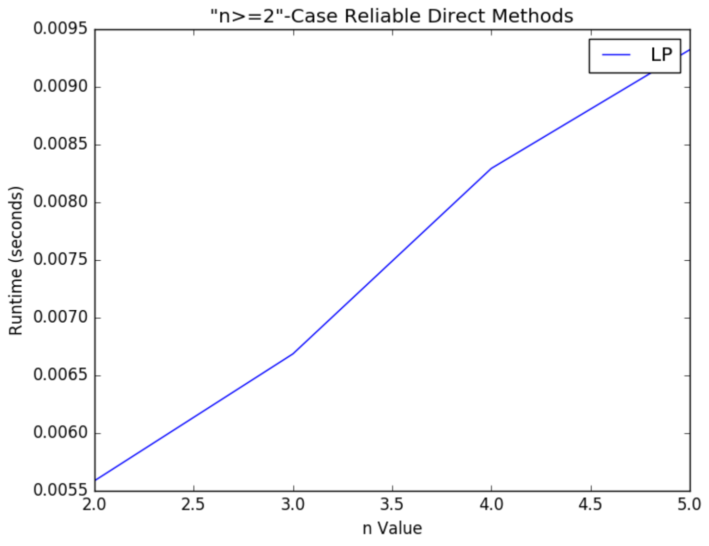

Figure 4: LP real-time runtime

8.3 Methods for Generating $\mathbb{P}^{(M)}$

In this section, we list the worst-case runtime complexities for all methods of generating $\mathbb{P}^{(M)}$. We then show their real-time performances for various inputs.

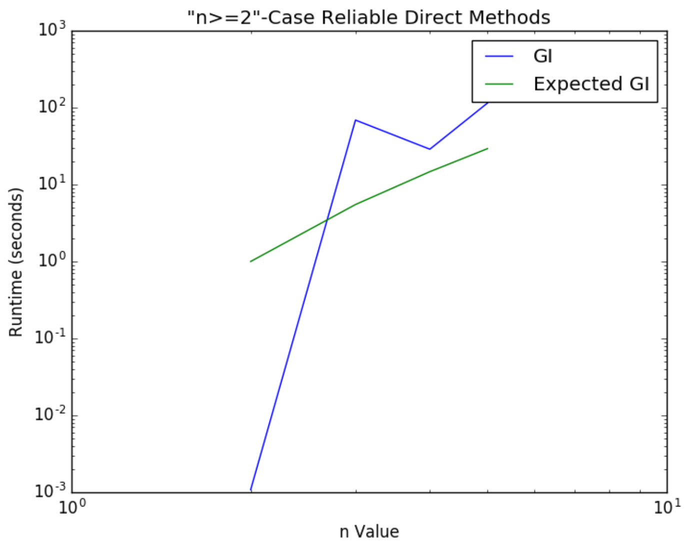

Figure 5: GI and Expected GI real-time runtime (logy vs. logx)

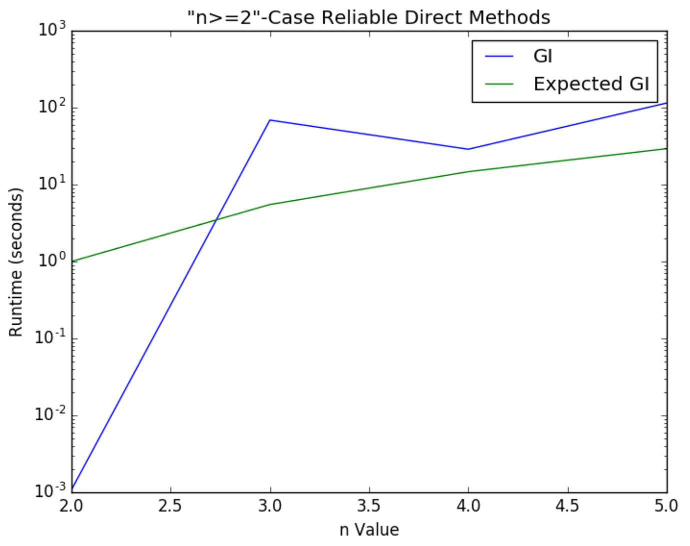

Figure 6: GI and Expected GI real-time runtime (logy vs. x)

8.3.1 Complexities

Here, we list the worst-case runtime complexities for the $\mathbb{P}^{(M)}$-generating methods. Again, all methods for generating $\mathbb{P}^{(M)}$ were tested on a computer with a single processor with values of $n \in \{2, 3, 4\}$, $N \in \{2, \dots, 30\}$. The VSEA quickly explodes in runtime from a matter of seconds to hours for increasing n and N. The Series Method tends to always take matters of seconds, but it was not tested for very large values of n or N.

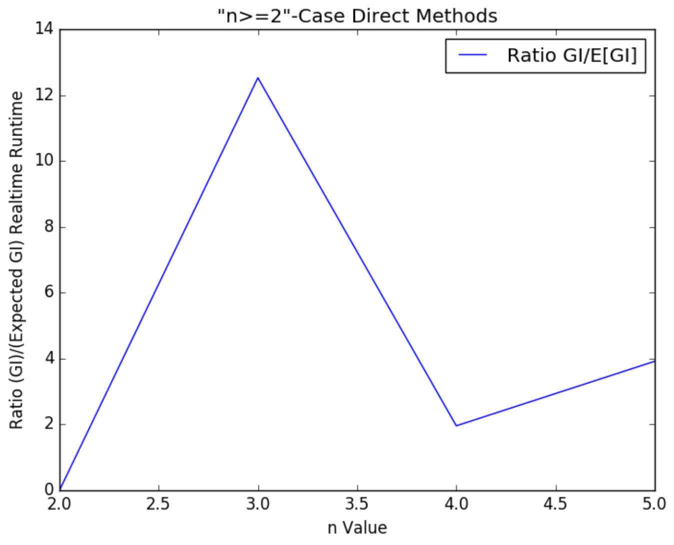

Figure 7: GI/Expected GI Ratio

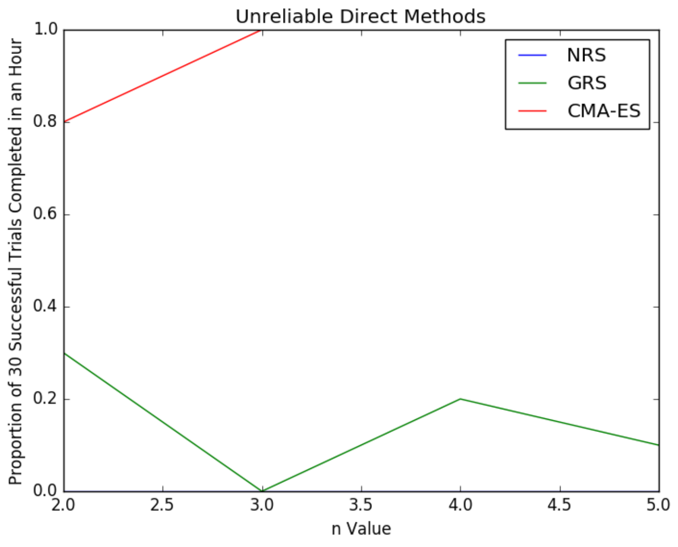

Figure 8: Proportion of trials the unreliable methods (NRS, GRS, CMA-ES) returned a solution in $60n$ seconds

- VSEA - $O\left(\binom{N+n-1}{n-1}^2\binom{2N+n!-1}{N+n!-1}(n!+N)\right)$
- Series Method - $O(Q(N,n)\cdot G(N,n)\cdot n^2\cdot\binom{N+n-1}{n-1}^2)$ where $Q(N,n)$ is the total number of successively bound weak compositions supported by v and resisted by w where $(v,w)\in \texttt{WC}(N,n)\times\texttt{WC}(N,n)$ $G(N,n)$ is the worst-case complexity in generating $\texttt{SBWC}(s^{(1)},s^{(2)})$.

8.3.2 Real-Time Runtimes

Here, we list graphs showing the real-time runtimes for the $\mathbb{P}^{(M)}$-generating methods for various (n, N)-tuples values.

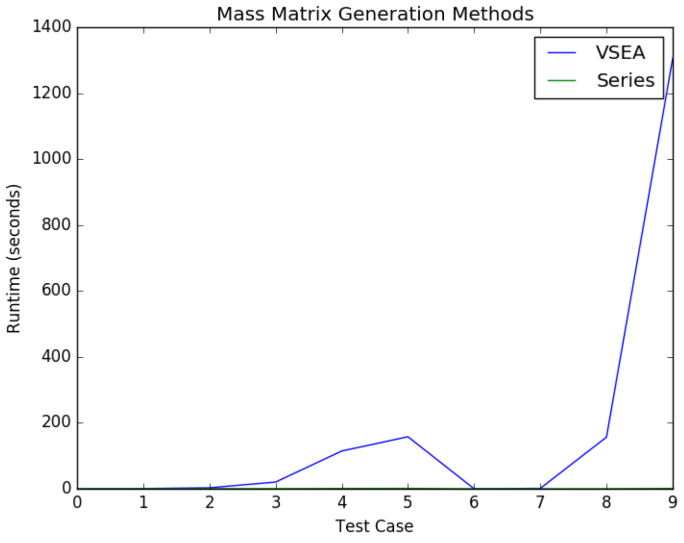

Figure 9: VSEA and Series plot

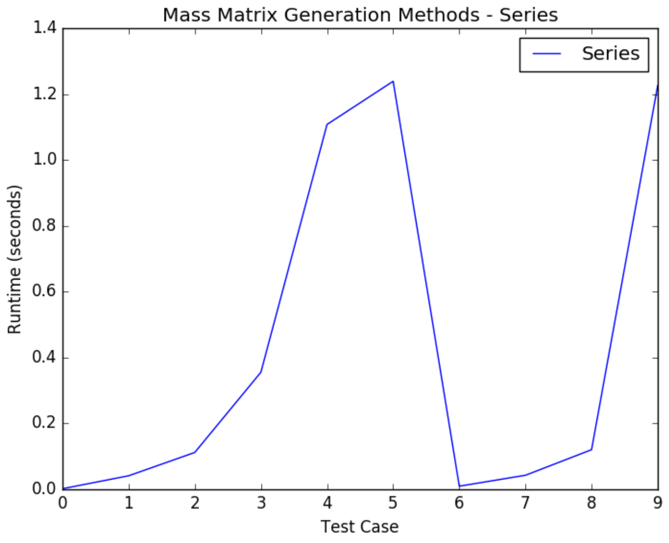

Figure 10: Series only plot

The test cases were the following:

Test Case	n	N	VSEA (s)	Series (s)
1	2	2	2.51793861e-03	0.01140809
2	10	2	6.66585207e-01	0.03999686
3	14	2	2.98541307e+00	0.09832096
4	21	2	2.03708050e+01	0.31426907
5	30	2	1.14650161e+02	0.93162298
6	32	2	1.57767011e+02	1.14047718
7	2	3	3.02648544e-02	0.0084362
8	2	4	1.26263905e+00	0.03489685
9	2	5	1.56787947e+02	0.11156487
10	4	4	1.30491586e+03	1.10382104

For larger n, N, both methods tend to quickly explode in real-time runtime.

9 Applications

Why may we need to obtain a $\mathbb{P}^{(I)}$ from a given π? This section answers that question by example. In general, applications require the following specifications: "Setting, Individuals, Bins, Welfare Function." The "setting" is the context within which individuals move about and the bins are intrinsic to the setting. The "individuals" are the actors who we categorize into some set of bins. The "bins" are the classes or categories intrinsic to the setting. Moreover, the distribution of individuals among bins is assumed to contribute, in some arbitrary way, to the "welfare function," which we seek to maximize or minimize. The determination of π, which is to occur before any algorithm developed in this paper is applied, is assumed to be the ideal distribution of individuals among all bins that maximizes or minimizes the welfare function. This distribution may be found through *integer programming* or some other optimization procedure.

For all applications, we will continue to hold the Identityless and Independence Assumptions for simplicity's sake, but we are in no way suggesting that these assumptions hold realistic merit and encourage further research to explore their relaxations.

9.1 Society, Citizens, Income, GDP/Welfare

Consider a society with individual citizens distributed among income classes. Perhaps its citizens intend to maximize its GDP or their overall welfare. Perhaps they also know how they would distribute themselves among income classes to optimize one or both objective functions. How do they alter public policy to converge to the optimal distribution? Apply any one of the previously mentioned direct methods of calculating $\mathbb{P}^{(I)}$. It ought to be noted that this situation is similar to the context "Company, Employees, Wage, Profit," where a company seeks to maximize its profit by optimally distributing its employees among wage brackets.

What is the significance of the Identityless and Independence Assumptions in this context? The former implies that all individuals experience the same impact of the same "policies" $\mathbb{P}^{(I)}$ whereas the latter implies that the experiences of individuals are unaffected by others. This is not necessarily how policies and people actually operate.

9.2 National Economy, Citizens, Rich-Poor, Wealth

This subsection presents an application to sustainable inequality management. The normative models presented by Galor and Zeira (1992) and the empirical evidence collected and corroborated by Kravis (1960), Lydall (1968) and the World Bank (1988, 1989, 1990, 1991) suggest that equity of income is correlated with income per capita. Tabellini (1990) also corroborates this claim but goes further to say that equity is also correlated with the rate of economic growth. [GZ93]

Clearly, there exists an incentive for any nation to distribute income equally among its citizens i.e. for $\forall i \in \{1 \ldots n\}; \pi_i = \frac{1}{n}$, and we know from the previous section that we can apply any of the direct methods to find the optimal set of policies $\mathbb{P}^{(I)}$ to guarantee convergence of a population to some π in a "Society, Citizens, Income, GDP/Welfare" context. Furthermore, we may borrow, from Galor and Zeira, the 2-bin classification of individuals: the skilled workers who invest their wealth in human capital and the unskilled workers who largely borrow wealth from the former class. Our problem now is much clearer: Find a $\mathbb{P}^{(I)}$ given that $n = 2, \pi^T = [\frac{1}{2}, \frac{1}{2}]$. Methods as simple as rS, bS, and brS may even be applied to this end.

Of course many solutions may be calculated. To demonstrate these algorithms in action, rS was selected to run six times, and for the input `rS(2, np.array([0.5,0.5]))` the following output was returned:

$$\begin{bmatrix} 0.9724358 & 0.0275642 \\ 0.0275642 & 0.9724358 \end{bmatrix}, \begin{bmatrix} 0.36960558 & 0.63039442 \\ 0.63039442 & 0.36960558 \end{bmatrix}, \begin{bmatrix} 0.4378469 & 0.5621531 \\ 0.5621531 & 0.4378469 \end{bmatrix}$$

$$\begin{bmatrix} 0.65705714 & 0.34294286 \\ 0.34294286 & 0.65705714 \end{bmatrix}, \begin{bmatrix} 0.75512384 & 0.24487616 \\ 0.24487616 & 0.75512384 \end{bmatrix}, \begin{bmatrix} 0.32285924 & 0.67714076 \\ 0.67714076 & 0.32285924 \end{bmatrix}$$

Call this output set A. Clearly A holds many different values for $\mathbb{P}^{(I)}$, but we should choose the value that is most similar to our current $\mathbb{P}^{(I)}$. This will minimize the amount of policy changes that need to be enacted and enforced to prompt society's convergence toward optimal wealth. Three methods are provided below to aid in finding a society's current $\mathbb{P}^{(I)}$ given their current assortment of policies.

9.2.1 Appeal to Regularization(s)

First, one may appeal to any regularization – choose a proposed $\mathbb{P}^{(I)}$ with the largest eigengap or trace, as discussed earlier. For situations involving larger $n > 2$, this approach may be impractical, because generating potential $\mathbb{P}^{(I)}$ may be costly (especially if some property of $\mathbb{P}^{(M)}$ is exploited) and values for $\mathbb{P}^{(I)}$ that maximize some regularization are found through an exhaustive search.

9.2.2 Long-Run Assertion

Alternatively, one may decide to assume their nation already exists in the "long-run," in which case their current distribution of "skilled" and "unskilled" people, as defined earlier, is taken to be π. Direct methods can then be applied thereafter to generate $\mathbb{P}^{(I)}$. These generated $\mathbb{P}^{(I)}$ may all have characteristics similar to only a subset of A. The issue is that the society may not actually be in the "long-run" – public policies that were recently enacted may be forcing the society to tend toward a different π.

9.2.3 Estimate Current $\mathbb{P}^{(I)}$

Finally, one may choose to estimate their current $\mathbb{P}^{(I)}$ directly from available, updated census data. For example, we can find an *maximum likelihood estimator (MLE)* for each element in the current $\mathbb{P}^{(I)}$. Fix one time step to be amount of time between any 2 census surveys. Organize the census data such that $X = (x_i)_{i=1}^{D}$ where

$$\forall i \in \{1 \ldots D\}; x_i = \left\{ \begin{array}{ll} [1,0,0,0] & \text{if person } i \text{ remained "skilled"} \\ [0,1,0,0] & \text{if person } i \text{ moved from "unskilled" to "skilled"} \\ [0,0,1,0] & \text{if person } i \text{ moved from "skilled" to "unskilled"} \\ [0,0,0,1] & \text{if person } i \text{ remained "unskilled"} \end{array} \right\}$$

In the absence of the Long-Run Assertion, we would have the following optimization problem to solve:

$$\begin{aligned}
\theta^* = [p_{11}^*, p_{22}^*] &= \text{argmax}_{\theta\in[0,1]^2} L(\theta|X) = \text{argmax}_{\theta\in[0,1]^2} P(X|\theta) \\
&= \text{argmax}_{\theta\in[0,1]^2} \prod_{i=1}^{D} P(x_i|\theta) \\
&= \text{argmax}_{\theta\in[0,1]^2} \sum_{i=1}^{D} \log(P(x_i|\theta)) \\
&= \text{argmax}_{\theta\in[0,1]^2} \left[\sum_{i=1}^{D_1} \log(p_{11}) + \sum_{i=1}^{D_2} \log(1-p_{22}) + \sum_{i=1}^{D_3} \log(1-p_{11}) + \sum_{i=1}^{D_4} \log(p_{22}) \right] \\
&= \text{argmax}_{\theta\in[0,1]^2} \left[D_1 \log(p_{11}) + D_2 \log(1-p_{22}) + D_3 \log(1-p_{11}) + D_4 \log(p_{22}) \right]
\end{aligned}$$

where $D_1 + D_2 + D_3 + D_4 = D$. Taking the gradient of both sides with respect to the elements of $\mathbb{P}^{(I)}$ and setting the entire gradient to $\vec{0}$, the $1 \times 2n$ zero vector:

$$\begin{aligned}
\vec{0} = \nabla \log(P(X|\theta)) &= \left[\frac{D_1}{p_{11}} - \frac{D_3}{1-p_{11}}, \frac{D_4}{p_{22}} - \frac{D_2}{1-p_{22}} \right] \\
&\Rightarrow D_3 p_{11} = D_1(1-p_{11}) \text{ and } D_2 p_{22} = D_4(1-p_{22}) \\
&\Rightarrow D_1 = p_{11}(D_1 + D_3) \text{ and } D_4 = p_{22}(D_4 + D_2) \\
&\therefore p_{11}^* = \frac{D_1}{D_1 + D_3} \text{ and } p_{22}^* = \frac{D_4}{D_2 + D_4}
\end{aligned}$$

where $L(\theta|X)$ is the likelihood function for parameters θ and data X, D_1 is the number of data samples (people) that remain skilled across any time step, D_2 is the number of people that move from unskilled to skilled across any time step, D_3 is the number of people that move from skilled to unskilled across any time step, and D_4 is the number of people that remain unskilled across any time step.

If the Long-Run Assertion is used in conjunction with this estimation method, then we would begin just as we did before, by taking the logarithm of the MLE:

$$[p_{11}^*, p_{22}^*] = \text{argmax}_{\theta\in[0,1]^2} L(\theta|X) = \text{argmax}_{\theta\in[0,1]^2} \sum_{i=1}^{D} \log(P(x_i|\theta))$$

However, we need to impose a new constraint in light of the Long-Run Assertion; We need to enforce that we exist in the "long-run" of our current $\mathbb{P}^{(I)}$, which suggests that:

$$M(\theta)\pi = \pi$$

which is equivalent to:

$$\|\mathtt{M}(\theta)\pi - \pi\|^2 = 0$$

where

$$\mathtt{M} : [a, b] \mapsto \begin{bmatrix} a & (1-b) \\ (1-a) & b \end{bmatrix}$$

is our "current $\mathbb{P}^{(I)}$." So we modify our objective function to now be:

$$f(\theta, \lambda) = \sum_{i=1}^{D} \log(P(x_i|\theta)) - \lambda \|\mathtt{M}(\theta)\pi - \pi\|^2$$

But we also know that both Equation (1) and Constraint (2) must hold whenever a 2×2 transition matrix has stationary distribution π. In other words, we can write our entire $\mathbb{P}^{(I)}$ in terms of a single parameter, $p \coloneqq p_{22}$. Therefore, our optimization problem simplifies to the following form:

$$p^* = \text{argmax}_{p \in [1-\frac{\pi_1}{\pi_2}, 1]} \Big[\log(L(p|X)) - \|\mathtt{M}(p, \pi)\pi - \pi\|^2 \Big]$$

where

$$\mathtt{M} : [a, \pi] \mapsto \begin{bmatrix} 1 - (1-a)\frac{\pi_2}{\pi_1} & (1-a) \\ (1-a)\frac{\pi_2}{\pi_1} & a \end{bmatrix}$$

Let us discuss why the second term, $-\lambda\|\mathtt{M}(p, \pi)\pi - \pi\|^2$, exists as a consequence of the Long-Run Assertion. The matrix $\mathtt{M}(p, \pi)$ is the estimate of the current $\mathbb{P}^{(I)}$, and the Long-Run Assertion tells us that our economy of interest already exists in the long-run. Therefore, $\mathtt{M}(p, \pi)\pi = \pi$. We want to minimize the extent to which this equality is not true, hence why we penalize the entire objective function whenever the difference between $\mathtt{M}(p, \pi)\pi$ and π is nonzero. The difference between vectors is classically given by the L_2-norm. We multiply this difference by $-\lambda$ for some $\lambda > 0$ because, again, we seek to penalize the objective function, which we intend maximize, whenever this difference exists. The λ is the extent to which we care about minimizing this difference.

Notice how the Long-Run Assertion acts simply as a regularization term within our MLE calculation, weighted by λ, penalizing any deviation between $\mathtt{M}(p, \pi)\pi$ and π for some p. This type of regularization, utilizing the L_2-norm, is called *L_2 Regularization*.

We may elect to treat this as an application of the method of *Lagrange Multipliers*, with multiplier λ. A necessary condition for this Lagrange Multiplier problem is that the function $f(p, \lambda)$ has a critical point and this implies that $\frac{\partial}{\partial p}\log(L(p|X)) - \lambda\frac{\partial}{\partial p}\|\mathtt{M}(p, \pi)\pi - \pi\|^2 = 0$ and $\|\mathtt{M}(p, \pi)\pi - \pi\|^2 = 0$. For a crucial reason, we will analyze the latter condition and *then* solve for the former condition.

The latter condition:

$$\|\mathtt{M}(\theta)\pi - \pi\|^2 = (p_{11}\pi_1 + (1-p)\pi_2 - \pi_1)^2 + ((1-p_{11})\pi_1 + p\pi_2 - \pi_2)^2$$
$$\Rightarrow 0 = \|\mathtt{M}(\theta)\pi - \pi\|^2 \Rightarrow (p_{11}\pi_1 + (1-p)\pi_2 - \pi_1)^2 = -((1-p_{11})\pi_1 + p\pi_2 - \pi_2)^2$$

which leaves us with only 1 situation for which there exists a real-valued solution p, namely when:

$$p_{11}\pi_1 + (1-p)\pi_2 - \pi_1 = (1-p_{11})\pi_1 + p\pi_2 - \pi_2$$

we must enforce this, and the importance of this enforcement will soon become crucial.

The former condition and remembering that $p_{11} = 1 - (1-p)\frac{\pi_2}{\pi_1} \Rightarrow \frac{\partial}{\partial p}p_{11} = -\frac{\pi_2}{\pi_1}$, we have:

$$\frac{\partial}{\partial p}f(p, \lambda) = \frac{\partial}{\partial p}\Bigg[\sum_{i=1}^{D_1}\log(p_{11}) + \sum_{i=1}^{D_2}\log(1-p) + \sum_{i=1}^{D_3}\log(1-p_{11}) + \sum_{i=1}^{D_4}\log(p) - \lambda\|\mathtt{M}(p, \pi)\pi - \pi\|^2\Bigg]$$
$$= \frac{\partial}{\partial p}\big[D_1\log(p_{11}) + D_2\log(1-p_{22}) + D_3\log(1-p_{11}) + D_4\log(p_{22})$$

$$-\lambda(p_{11}\pi_1 + (1-p)\pi_2 - \pi_1)^2 - \lambda((1-p_{11})\pi_1 + p\pi_2 - \pi_2)^2\big]$$

$$= \frac{-D_1}{p_{11}}\frac{\pi_2}{\pi_1} + \frac{-D_2}{1-p} + \frac{D_3}{1-p_{11}}\frac{\pi_2}{\pi_1} + \frac{D_4}{p} - 2\lambda(p_{11}\pi_1 + (1-p)\pi_2 - \pi_1)(-2\pi_2) - 2\lambda((1-p_{11})\pi_1 + p\pi_2 - \pi_2)(2\pi_2)$$

$$= \frac{-D_1}{p_{11}}\frac{\pi_2}{\pi_1} + \frac{-D_2}{1-p} + \frac{D_3}{1-p_{11}}\frac{\pi_2}{\pi_1} + \frac{D_4}{p} - 4\pi_2\lambda\big(-p_{11}\pi_1 - (1-p)\pi_2 + \pi_1 + (1-p_{11})\pi_1 + p\pi_2 - \pi_2\big)$$

From "The latter condition," we know that the last term is simply 0. We are ready to now set $\frac{\partial}{\partial p}f = 0$ to solve for p^*:

$$\frac{\partial}{\partial p}f(p,\lambda) = 0 = \frac{-D_1}{p_{11}}\frac{\pi_2}{\pi_1} + \frac{-D_2}{1-p} + \frac{D_3}{1-p_{11}}\frac{\pi_2}{\pi_1} + \frac{D_4}{p} = \frac{D_1}{1-p-\frac{\pi_1}{\pi_2}} + \frac{D_3 - D_2}{1-p} + \frac{D_4}{p}$$

We expect there to exist a value for p that solves the above equation for various values of constants D_1, D_2, D_3, D_4, and π. We also expect there to be permutations of constant values for which a solution for p does not exist. Regardless, it may be impossible to write an explicit formula for p^* given the overconstrained nature of the problem – we not only subject p to Constraint (2), but to the Long-Run Assumption as well. The *concavity/convexity* of f is also difficult to determine, so algorithms such as *gradient descent* cannot immediately be used to find p^* (unless a specific domain of p for which f is concave/convex is found). Considering how small the search space $[1 - \frac{\pi_1}{\pi_2}, 1]$ is, an exhaustive search of all possible values of p to some digit precision d may be practical. Such a search would incur a worst-case complexity of $O(\frac{\pi_1}{\pi_2}F(d)10^d)$, where $F(d)$ is the cost of d-digit precision arithmetic.

9.3 Nation, Genomes, Income, Income-per-Capita

This section presents an application to optimal genetic diversity policy as prescribed by Oded Galor. Galor notes that management of the genetic diversity of a nation aims to balance genetic diversity's "negative effect on the cohesiveness of society" with its "positive effect on innovations." Genetic diversity tends to increase mistrust among a citizenry and the amount of civil conflicts, which both tend to lead to inefficiencies in a macroeconomy relative to its *production possibility frontier* (everything that a macroeconomy can produce while operating at its peak efficiency). However, genetic diversity also "fosters innovations expands the production possibilities." Clearly, striking an optimal balance is of concern to those that value their society and can enact relevant policy change. Methods of managing national genetic diversity include the management of immigration and emigration policies and social norms that determine genetic mixing rates. [Gal]

9.3.1 Calculating and Managing Diversity

To calculate genetic diversity, we use the Index of Genetic Diversity from the online appendix to the work of Quamrul Ashraf and Oded Galor [AG], which takes the following form:

$$\mathbb{E}[H^{i,j}] = \frac{\theta_i\mathbb{E}[H^i] + \theta_j\mathbb{E}[H^j]}{1 - F^{i,j}}$$

where N is the total population size, θ_i is the share of people of ethnicity i in a group of people, H^i is the *heterozygosity* (genetic diversity) of ethnicity of i, $H^{i,j}$ is the heterozygosity of a population consisting of $\theta_i N$ member of ethnicity i and $\theta_j N$ members of ethnicity j, $F^{i,j}$ is the *genetic distance* between ethnicities i, j and $i, j \in \{1, \dots, k\}$ when we consider k ethnicities. Oftentimes, regions and countries consist of more than 2 ethnicities, in which case, the procedure of calculating $\mathbb{E}[H^{i,j}]$ is applied recursively i.e. $\mathbb{E}[H^i] \coloneqq \mathbb{E}[H^j], \theta_i \coloneqq \theta_j, j \coloneqq j + 1$.

Unfortunately, $F^{i,j}$ values only exist for 53 ethnic groups (or pairs thereof), so instead of the above index, parametric estimates of $F^{i,j}$ and $\mathbb{E}[H^i]$ have been formulated. It has been empirically shown that $\mathbb{E}[H^i]$ is strongly, negatively correlated with d_i, the distance between that ethnicity's geographic location in 1 C.E. and East Africa (the alleged birthplace of humanity). Additionally, $F^{i,j}$ has been empirically shown to be strongly, positively correlated with d_ij, the *pairwise migratory distance* between ethnicities i and j (the geographic distance between these ethnicities' regions of origin). With this in mind, the following parametric index of genetic diversity is often employed:

$$\mathbb{E}[\hat{H}^{i,j}] = \frac{\theta_i \mathbb{E}[\hat{H}^i(d_i)|d_i] + \theta_j \mathbb{E}[\hat{H}^j(d_j)|d_j]}{1 - \hat{F}^{i,j}(d_j)}$$

If we were to use "country of origin" as a proxy for ethnicity, then i, j would parse over all indices possible countries of origin instead of indices of all ethnicities, and $\hat{H}^i(d_i)$ would be the heterozygosity of country i that is a distance d_i from East Africa.

Oded Galor *et al.* [Gal] already empirically found the "ideal," income-per-capita maximizing value for $\mathbb{E}[H^i]$, which is $\hat{H}^* = 0.2792$. For context, the U.S. is $\hat{H}^{U.S.} = 0.2794$, a tad more diverse than the empirically optimum expected heterozygosity [Gal]. Given $\mathbb{E}[\hat{H}^i]$ and $F^{i,j}, \forall i, j \in \{1, \ldots, k\}$, we can solve for a vector $\vec{\theta}^* = (\theta_1, \ldots, \theta_k)$ that minimizes $\hat{H}^* - \hat{H}^l$ for a country l with with immigrants and citizens hailing from a total of k different countries, enumerated as $\{1, \ldots, l, \ldots, k\}$. We can solve for $\vec{\theta}^*$ using *dynamic programming* (also called *memoization*), a method of optimizing recursively-defined functions that stores all potentially optimum values of subproblems in tables. In this case, there are k subproblems, which each solve for a $\theta_i, i \in \{1, \ldots, k\}$. Again, finding an ideal $\vec{\theta}^*$, which is related to our ideal π, is beyond the scope of this paper. In practice, upon finding $\vec{\theta}^*$, we will learn to what extent we should adjust our immigration or emigration rates for a particular nationality to better optimize our income-per-capita.

9.3.2 Genomic Diversity Model

We will again use "country of origin" as proxy for genetic diversity and assume we have access to $\mathbb{E}[\hat{H}^i]$ and $F^{i,j}$ for each country i and assert that there are k countries of origin that their citizens can either be found domestically or abroad. Thus, there are 2 bins per country of origin. Using $\mathbb{E}[\hat{H}^i]$ values per each i nation as opposed to each i ethnicity allows our model to permit genetic mixing within nations. We end up with multiple $\mathbb{P}_i^{(I)}$ and π_i, in particular one per each nation i. Our model is:

$$\forall i \in \{1, \ldots, k\}$$
$$\mathbb{P}_i^{(I)} = \begin{bmatrix} p_{dd}^{(i)} & p_{ad}^{(i)} \\ p_{da}^{(i)} & p_{aa}^{(i)} \end{bmatrix}$$
$$\pi_i = \begin{bmatrix} \theta_i \\ 1 - \theta_i \end{bmatrix}$$

Note how this situation is identical to the $n = 2$ case of Equation (4), which represents what occurs upon relaxation of the Independence Assumption.

9.4 Generalization of Applications

In this section, we abstractly define the space of all possible applications of the methods discussed in this paper.

Define an *Application Space* to be any ordered pair (n, N, ρ) where $n, N \in \mathbb{N}, n \geq 2, N \geq 1$, and $\rho : U \subset \mathbb{R}^n \to \mathbb{R}_+$, where $\mathbb{R}_+$ are the nonnegative real numbers and U is the set of all elements of $\texttt{WC}(N, n)$, each normalized to 1. Let ρ be a *preference metric* – our means of rating all π we could possibly seek given parameters n and N. The more we wish to converge to a particular π, the larger the value of $\rho(\pi)$. More concretely:

$$\forall \vec{x}, \vec{y} \in U; \vec{x} \prec \vec{y} \Rightarrow \rho(\vec{x}) < \rho(\vec{y})$$

where $\vec{x} \prec \vec{y}$ means "we prefer $\vec{y}$ more than $\vec{x}$." Examples of preference functions may be GDP – a nation's GDP may vary as the distribution of its people among income classes varies, so we prefer population distributions associated with a greater GDP value. If we were to enumerate all of these possible π_i and sum them with weights $\rho(\pi_i)$, we would arrive at the π to which we seek to converge. In other words, if $s = \binom{N+n-1}{n-1}$,

$$\pi = \sum_{i=1}^{s} \rho(\pi_i)\pi_i$$

We can then find a $\mathbb{P}^{(I)}$ that converges to this π using the previously mentioned methods. Note how this summation over all possible π_i is the same as summing over all elements of the first eigenvector of $\mathbb{P}^{(M)}$, as described in Section (7.1).

10 Conclusion

10.1 Summary

In this paper, we worked backwards relative to what is taught in most undergraduate courses concerning MCs; We learned the various ways of solving for a transition matrix given a stationary distribution. After motivating the problem, we defined all relevant definitions and then described several methods for directly solving for what we called the individual matrix, $\mathbb{P}^{(I)}$, namely: bS, rS, brS, LP, NRS, GRS, GI, and CMA-ES. Worst-case and expected time complexities were provided when available. Complexity calculations were provided for neither GRS nor CMA-ES. In our discussion on the CMA-ES, we took a detour into mixing times, second eigenvalues and why their relevant to regularizations. This detour was relevant to CMA-ES because this is the only method capable of handling "tricky" objective functions, which may exhibit non-linearity, are multimodal, are non-smooth, etc.

Our discussion then drifted toward learning what happens when our core assumptions – the Identityless and Independence Assumptions – were relaxed. This discussion transitioned into another discussion concerned with the generation of $\mathbb{P}^{(M)}$, the mass matrix. We learned about the greatly complicated and inefficient VSEA and how it inspired its speedy relative, the Series Method. Performances of each algorithm were provided when available. Toward the end of our discussion on the Series Method, we had the necessary knowledge to learn more about how we may relax the Independence Assumption, providing two approaches: the Quick-and-Dirty Approach and the Formal Approach.

The complexities of all methods, when available, were listed in the "Summary of Methods and Performance" before we pivoted into learning about how any of these methods may be applied to real-world scenarios. Our applications were: "Society, Citizens, Income, GDP/Welfare," "Company, Employees, Wage, Profit," "National Economy, Citizens, Rich-Poor, Wealth" and " Nation, Genomes, Income, Wealth." We provided a clear means of applying the mathematical machinery developed earlier to all application contexts. We concluded our discussion of applications with a generalization of all possible applications.

10.2 Future Research

Future research should concern itself with the optimization of each algorithm's complexity, particularly the CMA-ES, as its exact runtime, given any of the mentioned regularizations or objective functions, is unknown to the author. Parallelization of methods other than GI should be explored. Other linear objective functions should be explored so LP can return more informative results. More efficient algorithms that can handle the "tricky" objective functions and regularizations mentioned in this paper should also be developed.

With specific regards to the generation of $\mathbb{P}^{(M)}$, connections should be drawn to the field of analytic combinatorics. The literature currently contains plenty of information regarding n-Chamber Ehrenfest Models, for small n, usually $n \in \{2, 3\}$, but what is needed to potentially optimize the runtime complexity of generating $\mathbb{P}^{(M)}$ is research regarding n-Chamber Completely-Connected Ehrenfest Models, where $n > 3$, all n chambers are connected to one another, and no particle is forced to move between chambers between time steps with some probability. Perhaps the runtime of generating $\mathbb{P}^{(M)}$ can be lessened to an even greater extent by abstracting the space of population sizes (values for N) away from solely discrete spaces, into the continuum.

11 Appendix

This section lists all algorithms mentioned in this paper, coded in Python 2.7.13. All code can be accessed online on GitHub at:

https://github.com/kpeluso/thesis/tree/master/Clean

11.1 bS

```python
from HELPERS_clean import TOL, np, listMatch, resMat, corrEv, p22p1

def bS(n, pi, iters=float('inf'), tol=TOL):
        '''
        INPUT:
                N :: Integer
                        # population size
                n :: Integer
                        # number of bins
                pi :: NPArray<Float>
                        # optimal stationary distribution
        OUTPUT:
                NPArray<NPArray<Float>>
                        # P^(I) via Binary Search Method
                        # only works in 2x2 case
                        # BECAUSE IT'S 2x2 CASE I CAN USE 2x2 P^(M) SERIES FORMULA
        '''
        output = np.ones([2,2])*0.5 # P^(I)
        ev = corrEv(output)
        b1 = 0.0 if pi[0] >= pi[1] else 1.0-(pi[0]/pi[1])
        b2 = 1.0
        output[1][1] = np.average([b1,b2])
        output = resMat(output)
        while not listMatch(np.dot(output, ev), pi, tol=tol) and iters > 0: # s1, loop
                if ev[1] < pi[1]:
                        b1 = np.average([b1,b2])
                        output[1][1] += np.average([b1,b2])-b1
                else:
                        b2 = np.average([b1,b2])
                        output[1][1] -= np.average([b1,b2])-b1
                output[0][0] = p22p1(output[1][1], pi)
                output = resMat(output)
                ev = corrEv(output)
                iters -= 1
        return output
```

11.2 rS

```
from HELPERS_clean import TOL, np, resMat, corrEv, p22p1, genStat, listMatch

def rS(n, pi, iters=float('inf'), tol=TOL):
        '''
        INPUT:
                N :: Integer
                        # population size
                n :: Integer
                        # number of bins
                pi :: NPArray<Float>
                        # optimal stationary distribution
        OUTPUT:
                NPArray<NPArray<Float>>
                        # P^(I) via Random Search Method
                        # only works in 2x2 case
                        # BECAUSE IT'S 2x2 CASE I CAN USE 2x2 P^(M) SERIES FORMULA
        '''
        output = np.zeros([2,2]) # P^(I)
        for col in xrange(2):
                output[:,col] = np.transpose(genStat(2))
        ev = corrEv(output)
        b1 = 0.0 if pi[0] >= pi[1] else 1.0-(pi[0]/pi[1])
        b2 = 1.0
        output[1][1] = np.average([b1,b2])
        output = resMat(output)
        while not listMatch(np.dot(output, ev), pi, tol=tol) and iters > 0: # s1, loop
                output[1][1] = (b2-b1)*np.random.random()+b1
                output[0][0] = p22p1(output[1][1], pi) # calculate p_11
                output = resMat(output) # calculate p_12, p_21
                ev = corrEv(output)
                iters -= 1
        return output
```

11.3 brS

```
from HELPERS_clean import TOL, np, listMatch, resMat, corrEv, p22p1, next_term

def brS(n, pi, iters=float('inf'), tol=TOL):
        '''
        INPUT:
                N :: Integer
                        # population size
                n :: Integer
                        # number of bins
                pi :: NPArray<Float>
                        # optimal stationary distribution
        OUTPUT:
                NPArray<NPArray<Float>>
                        # P^(I) via Binary-Random Search Method
                        # only works in 2x2 case
                        # BECAUSE IT'S 2x2 CASE I CAN USE 2x2 P^(M) SERIES FORMULA
        '''
        output = np.ones([2,2])*.5 # P^(I)
        ev = corrEv(output)
        b1 = 0.0 if pi[0] >= pi[1] else 1.0-(pi[0]/pi[1])
        b2 = 1.0
        output[1][1] = np.average([b1,b2])
        output = resMat(output)
        while not listMatch(np.dot(output, ev), pi, tol=tol) and iters > 0: # s1, loop
                # look for random number within updated b1, b2
                        if ev[1] < pi[1]:
                                b1 = np.average([b1,b2])
                                output[1][1] += next_term(b1,b2)
                        else:
                                b2 = np.average([b1,b2])
                                output[1][1] -= next_term(b1,b2)
                        output[0][0] = p22p1(output[1][1], pi)
                        output = resMat(output)
                        ev = corrEv(output)
                        iters -= 1
        return output
```

11.4 LP

```
import numpy as np
from HELPERS_clean import TOL, devectorize, normize, listMatch
from scipy.optimize import linprog

def LP(n, pi, tol=TOL):
    '''
        INPUT:
                n :: Integer
                        # number of bins
                pi :: NPArray<Float>
                        # optimal stationary distribution
        OUTPUT:
                NPArray<NPArray<Float>>
                        # P^(I) via LP Method, solved with a black box Simplex Method
     '''
    # build objective function - trace(P^(I))
    c = []
    for i in xrange(n):
        c += [0]*i + [1] + [0]*(n-1-i)
    c = -1*np.array(c) # default is to minimize, so we multiply -1 to maximize
    # constraint 1 - InnerProduct(P^(I)[j,:], pi) == pi[j] for all j=1:n
    M = np.zeros([2*n,n**2])
    pi_l = list(pi)
    for i in xrange(n):
        M[i,:] += np.array([0]*(i*n) + pi_l + [0]*(n**2-i*n-n))
    # constraint 2 - Sum(P^(I)[:,j]) == 1 for all j=1:n
    for i in xrange(n,2*n):
        M[i,:] += np.array([0]*((i-n)*n) + [1]*n + [0]*(n**2-(i-n)*n-n))
    # constraint 3 - P^(I)[i,j] >= 0 for all i,j=1:n (nonnegativity constraints)
    #   0<p<1 for all parameters since we don't want Identity Matrix as output
    bounds = [(TOL,1.0-TOL)]*n**2
    # Mx = b
    b = pi_l + [1]*n
    output = linprog(c, A_ub=M, b_ub=b, bounds=bounds, method='interior-point')
    # print '\noutput', output
    # print 'output[x]', output['x']
    # print len(c), c
    # print len(M), M
    # print len(b), b
    # print len(bounds), bounds
    # print ' '
    # print '\nMatch?:', listMatch(np.dot(normize(devectorize(list(output['x']), n)), pi), pi, tol=T
    return normize(devectorize(output['x'], n))
```

11.5 NRS

```
from HELPERS_clean import TOL, np, listMatch

def NRS(n, pi, iterCols=float('inf'), iterInCol=10000, tol=TOL):
        '''
        INPUT:
                n :: Integer
                        # number of bins
                pi :: NPArray<Float>
                        # optimal stationary distribution
        OUTPUT:
                NPArray<NPArray<Float>>
                        # P^(I) via Naive Random Search Method
        '''
        # initialization and normalization
        if iterCols == None:
                iterCols = n*10000
        output = np.random.random((n,n))
        for col in xrange(n):
                output[:,col] /= sum(output[:,col])
        out = np.linalg.eig(output)
        n0s = np.zeros(n)
        totalIts = iterCols; totalItsIn = iterInCol
        while (not abs(out[0][0] - 1.0) < tol or not listMatch(out[1][:,0], pi)) and iterCols!=0:
                for col in xrange(n):
                        p_old = n0s
                        iterInCol = totalItsIn
                        temp = output
                        while not listMatch(output[:,col], p_old) and iterInCol!=0:
                                next_try = np.random.random((n,))
                                next_try /= sum(next_try) # Line 5 in LaTeX
                                temp[:,col] = next_try
                                out = np.linalg.eig(temp)
                                if np.linalg.norm(out[1][:,0] - pi)**2 < np.linalg.norm(p_old - pi)**
                                        output = temp
                                        p_old = output[:,col]
                                iterInCol -= 1
                out = np.linalg.eig(output)
                iterCols -= 1
        return output
```

11.6 GRS

```python
from HELPERS_clean import np, TOL, listMatch, corrEv

def GRS(n, pi, maxIters=float('inf'), tol=TOL):
        '''
        INPUT:
                N :: Integer
                        # population size
                n :: Integer
                        # number of bins
                pi :: NPArray<Float>
                        # optimal stationary distribution
        OUTPUT:
                NPArray<NPArray<Float>>
                        # P^(I) via Greedy Random Search Method
        '''
        # initialization and normalization
        if maxIters == None:
                maxIters = n*100000
        output = np.random.random((n,n))
        for col in xrange(n):
                output[:,col] /= sum(output[:,col])
        out = np.linalg.eig(output)
        ev = corrEv(out[1])
        while not listMatch(np.dot(output, pi), pi) and maxIters != 0: # Step 2
                a = np.random.randint(0, high=n) # Step 3
                c = 0.45*float((2*int(ev[a] < pi[a]) - 1)) # Step 4
                # Step 5
                output[a,a] = output[a,a]*(int((1.0+c)*output[a,a] > 1.0)+int(abs((1.0+c)*output[a,a]
                        + (1.0+c)*output[a,a]*int((1.0+c)*output[a,a] <= 1.0)*int((1.0+c)*output[a,a]
                output[:,a] /= sum(output[:,a]) # Step 6
                out = np.linalg.eig(output) # preparation for Step 2
                ev = corrEv(out[1]) # preparation for Step 2
                maxIters -= 1 # Step 7
        return output # Step 8
```

11.7 GI

```
from HELPERS_clean import TOL, np, corrEv, genStat, listMatch
from brS_clean import brS

def GI(n, pi, iters=float('inf'), tol=TOL):
        '''
        INPUT:
                n :: Integer
                        # number of bins
                pi :: NPArray<Float>
                        # optimal stationary distribution
                        # CAN'T HAVE ANY ZERO ENTRIES!
        OUTPUT:
                NPArray<NPArray<Float>>
                        # P^(I) via Gibbs Sampling-inspired Method
        '''
        output = np.zeros([n,n]) # P^(I)
        for col in xrange(n):
                output[:,col] = np.transpose(genStat(n))
        ev = corrEv(output)
        indices = range(n)
        while not listMatch(np.dot(output, ev), pi) and iters > 0: # s1, loop
                # s2, isolate
                alter = np.random.choice(indices, size=[2], replace=False).astype(int)
                alter = np.array([min(alter), max(alter)]) # sort in order of lowest to highest
                subpi = np.zeros(2)
                subpi[0] = pi[alter[0]]
                subpi[1] = pi[alter[1]]
                # s3b, note how much space was formerly taken up
                resMass_mat = (output[alter[0]][alter[0]] + output[alter[1]][alter[0]], \
                        output[alter[0]][alter[1]] + output[alter[1]][alter[1]])
                resMass_pi = sum(subpi)
                # s3, normalize
                subpi /= sum(subpi)
                # s4, optimize extracted 2-equation system
                submat = brS(n, subpi) # !!! Use bS, rS, brS methods. !!!
                # s5a, denormalize
                submat[:,0] *= resMass_mat[0]
                submat[:,1] *= resMass_mat[1]
                subpi *= resMass_pi
                # s5, substitute in new values renormalized to Q
                output[alter[0]][alter[0]] = submat[0][0]
                output[alter[1]][alter[0]] = submat[1][0]
                output[alter[0]][alter[1]] = submat[0][1]
                output[alter[1]][alter[1]] = submat[1][1]
                ev = corrEv(output)
                iters -= 1
        return output
```

11.8 CMA-ES

```
from HELPERS_clean import TOL, np, devectorize, stochTest
import cma


def anyNeg(l):
        for el in l:
                if el < 0.0:
                        return True
        return False


def corrEig(vm, corr, bins, tol=10.0*TOL):
        '''
        INPUT:
                vm :: NPArray<Float>
                        # bins^2-length vectorized P^I
                corr :: NPArray<Float>
                        # bins-length correct stationary distribution
                bins :: Integer
                        # the number of bins
                tol :: Float
                        # tolerance
        OUTPUT:
                Float
                        # The error between corr and the first eigenvector of vm
                        # If first eigenvalue is not within tol of 1.0, then np.NaN is returned
        '''
        mat = devectorize(vm, bins)
        out = np.linalg.eig(mat)
        eigenvalue1 = out[0][0]
        # if first eigenvalue is not 1.0 or columns not probability distributions (between 0-1 and su
        if not abs(eigenvalue1 - 1.0) < tol or anyNeg(vm) or not stochTest(mat, tol=tol):
                return np.NaN
        else:
                eigenvector1 = out[1][:,0]
                return sum(abs(corr - eigenvector1)) # 1-D loss
                # return corr - eigenvector1 # (bins)-D loss


def CMAES(n, pi, tol=10.0*TOL):
        # build x0
        x0 = np.zeros(n**2)
        for i in range(n):
                nxt = np.random.uniform(0,1,n)
                x0[i*n:i*n+n] = nxt/sum(nxt)
        # build sigma0
        sigma0 = .25**n # ~1/4 of search domain width => try with .25 and (.25)**n
        f = cma.fitness_functions.FitnessFunctions()
        preFit = lambda x: corrEig(x, pi, n, tol)
        fitnessFunction = lambda x: f.fun_as_arg(x, preFit)
        output = cma.fmin(fitnessFunction, x0, sigma0)
        print output
        print '\n'
        return devectorize(output[5], n) # incumbent solution found at this entry in output array
```

11.9 VSEA

```
from HELPERS_clean import TOL, np, weak_compositions, permute, perm, listMatch
from scipy.special import comb
from scipy.misc import factorial
from copy import deepcopy


def buildVectList(size, mat):
        '''
        INPUT:
                size :: Integer
                        # n
                mat :: List<List<Integer>>
                        # individual matrix
        OUTPUT:
                List< Tuple< List<Integer>, Float > >
                        # first index is permutation of (n-2)*0,1,-1, next is P[source -> sink]
        '''
        output = []
        for perm in iter(permute([1,-1]+[0]*(size-2))):
                oneIdx = perm.index(1); negOneIdx = perm.index(-1)
                output.append((np.array(perm), mat[oneIdx][negOneIdx]))
        return output


def allotCombos(st, leavs, ents):
        '''
        INPUTS:
                st :: List<Integer>
                        # number of people in each bin in starting state (Tj)
                leavs :: List<Ineger>
                        # N-length list of people leaving each bin
                ents :: dict<Integer:List<Integer>>
                        # key=source bin : value=N-length list of numbers of unit sinks per bin
        OUTPUT:
                Integer
                        # total number of ways of sorting people into bins given variable, known spac
        '''
        output = 1
        for idx, bi in enumerate(st):
                output *= comb(bi, leavs[idx])
                if bi > 0 and output > 0 and leavs[idx] > 0:
                        sinked = 0 # number of people already allotted
                        for bi2 in ents[idx]:
                                output *= comb(leavs[idx]-sinked, bi2)
                                sinked += bi2
        return output


def VSEA(N, n, indMC): # non-unique, identity-less individuals
```

```
        '''
        INPUT:
                N :: Integer
                        # number of people
                n :: Integer
                        # number of bins
                indMC :: NPArray<NPArray<Float>>
                        # individual matrix
        OUTPUT:
                List<List<Float>>
                        # mass matrix
        '''
        #
        # Block A
        #
        sl = int(comb(N+n-1, n-1)) # side length
        output = np.zeros([sl, sl])
        TESTLIST = []
        vectList = buildVectList(n, indMC) # lacks 0 vectors
        zeroVect = np.zeros(n)
        #
        # Block B
        #
        i = 0 # current row index
        for Ti in weak_compositions(N, n): # rows of mass matrix
                print 'Progress:', 100.0*float(sl*i)/float(sl**2), '%'
                j = 0 # current column index
                for Tj in weak_compositions(N, n): # columns of mass matrix
                        #
                        # Block C
                        #
                        diff = np.array(Tj) - np.array(Ti) # Prospective state transition: Tj -> Ti
                        vectList_copy = deepcopy(vectList)
                        # add 0 vectors to vectList
                        max_0s_per_bin = []
                        for bn in xrange(n):
                                max_0s_per_bin.append(min(Tj[bn], Ti[bn]))
                                if Tj[bn]>0 and Ti[bn]>0:
                                        vectList_copy.append((np.zeros(n), indMC[bn][bn], bn))
                        #
                        # Block D
                        #
                        # get every way of allotting N tokens across all vectors in vectList
                        lvlc = len(vectList_copy)
                        transition_prob_terms = []
                        for wc in weak_compositions(N, lvlc): # N = number of tokens, lvlc = number 
                                sum_vect = np.zeros(n)
                                breakIt = False
                                static_in_bin = np.zeros(n) # number of people staying in their bin
                                possibly_correct_vects = [] # :: List<Tuple< vector, prob, number_of
# idx0 if 0 vect, gives corresponding diagonal index of particular 0 vect
```

```
                for vectTup_idx in xrange(lvlc):
                    if wc[vectTup_idx] > 0: #check if vector is used
                        # check if too many of the same zero vector has been
                        if listMatch(vectList_copy[vectTup_idx][0], zeroVect)
                            static_in_bin[vectList_copy[vectTup_idx][2]]
                            if max_0s_per_bin[vectList_copy[vectTup_idx][
                                breakIt = True # too many of the same
                                break
                        sum_vect += float(wc[vectTup_idx])*vectList_copy[vect
                        possibly_correct_vects.append((vectList_copy[vectTup_
                if breakIt: # skip to next wc
                    continue
                #
                # Block E
                #
                if listMatch(diff, sum_vect): # check if current weak composition wc
                    # coefficient determination
                    leaving = np.zeros(n); entering = {}; prod = 1
                    for pre_pvs in possibly_correct_vects:
                        pvs = np.ndarray.tolist(pre_pvs[0])
                        if not listMatch(pvs, zeroVect): # when we DON'T enco
                            for bi in xrange(n):
                                if pvs[bi] == -1:
                                    leaving[bi] += pre_pvs[2]
                                    if bi not in entering:
                                        entering[bi] = np.zer
                                    entering[bi][pvs.index(1)] +=
                                    break # jump to next pre_pvs
                    # check if too many people are leaving a bin <— REALLY HACKY
                    for bi in xrange(n):
                        if leaving[bi] + static_in_bin[bi] > Ti[bi]:
                            breakIt = True
                            break
                    if breakIt: # skip to next wc
                        continue
                    #
                    # Block F
                    #
                    prod *= allotCombos(Ti, leaving, entering) # get coefficient
                    # get P^I probabilities that correspond to selected basis ve
                    #   get exponents for all probabilities that correspond to th
                    #   quantities of each of their respective basis vector
                    pvstEST = 1
                    for pvs in possibly_correct_vects:
                        prod *= pvs[1]**pvs[2]
                        pvstEST *= pvs[1]**pvs[2]
                    transition_prob_terms.append(prod) # this is just one term th
            output[j][i] = sum(transition_prob_terms); j += 1 # add element to mass matr
        i += 1
    return output
```

11.10 Series Method

```
from HELPERS_clean import np, listMatch, stochTest, wc_count, weak_compositions, comb
from copy import deepcopy

def bounded_wcs(balls, boxes, minBalls, maxBalls, currBox=0, parent=tuple(), first=True, iters=0):
        '''
        Each bin i in a given generated output must have at least minBalls[i] balls
                but no more than maxBalls[i] balls.

        sum(minBalls) <= balls <= sum(maxBalls)

        currBox indexed at 0
        '''
        #num_funct_calls.append(iters)
        if first and sum(minBalls) > balls:
                print '\nError! :: Too few balls given the constraint minBalls!\n'; quit()
        if first and sum(maxBalls) < balls:
                print '\nError! :: Too many balls given the constraint maxBalls!\n'; quit()
        # calculate degrees of freedom and only vary within them
        #       (^unless this is already optimal)
        if balls > sum(maxBalls[currBox:]):
                pass
        else:
                if boxes > 1:
                        i = max(0, balls-maxBalls[currBox]) # balls leftover after currBox
                        while i <= balls-minBalls[currBox]:
                                for x in bounded_wcs(i, boxes-1, minBalls, maxBalls, currBox+1, parer
                                        yield x
                                i+=1
                else:
                        yield parent + (balls,)


def mn(n, k):
        '''
        INPUT:
                n :: Integer
                k :: List<Integer>
                        # sum(k) <= n
        OUTPUT:
                Integer
                        # multinomial choose calculator i.e. multichoose
        '''
        result = 1
        less = 0
        for i in k:
                result *= comb(n-less,i)
                less += i
        return result
```

```
def p_wc(ls, n, initial_maxs, initial_output=[], currIdx=0):
        '''
        INPUT:
                ls :: NPArray<Integer>
                n :: Integer
                        # len(ls)
                initial_maxs :: NPArray<Integer>
                        # should have length len(ls)
        OUTPUT:
                generated List<NPArray<Integer>>
                        # a successively bounded wc list as generated
        '''
        n0s = np.zeros(n)
        if currIdx == n or ls == [] or listMatch(initial_maxs, n0s):
                yield initial_output
        else:
                for bwc in bounded_wcs(ls[currIdx], n, n0s, initial_maxs):
                        next_wc = np.array(bwc).astype(int)
                        i_o = deepcopy(initial_output)
                        i_o.append(next_wc)
                        for x in p_wc(ls, n, initial_maxs-next_wc, i_o, currIdx+1):
                                yield x


def Series(N, n, indMat):
        '''
        INPUT:
                N :: Integer
                        # number of people
                n :: Integer
                        # number of bins
                indMat :: List<List<Float>>
                        # individual matrix
        OUTPUT:
                List<List<Float>>
                        # mass matrix according to my generalized series formula (without Java speedu
        '''
        sl = wc_count(N,n) # side length
        print 'Progress:'
        output = np.zeros([sl, sl])
        n0s = np.zeros([n, n])
        i = 0 # current row index
        for Ti in weak_compositions(N, n): # rows of mass matrix (s^(2))
                j = 0 # current column index
                for Tj in weak_compositions(N, n): # columns of mass matrix (s^(1))

                        b1 = 0 # block 1
                        for scwc in p_wc(np.array(Tj), n, np.array(Ti)): # loop over successively-co
                                b2 = 1 # block 2
                                for d_idx, donor in enumerate(Tj):
                                        if donor > 0:
                                                b2 *= mn(donor, scwc[d_idx])
                                                for t in xrange(n): # block 3
                                                        b2 *= indMat[t][d_idx]**scwc[d_idx][t]
                                b1 += b2

                        output[i][j] = b1
                        j += 1
                i += 1
                print 100.0*float(i)/float(sl), '%' # progress output
        return output
```

11.11 Helper Functions

```
import numpy as np
from scipy.special import comb

TOL = 0.01

p22p1 = lambda p2,pi: 1.0 - (1.0 - p2)*(pi[1]/pi[0]) # proof of formula given in rough draft of thes

vect = lambda mat: [mat[0][0], mat[1][0], mat[0][1], mat[1][1]]

next_term = lambda a,b: (b-a)*np.random.random() # b >= a

wc_count = lambda n,k: int(comb(n+k-1,k-1)) # count total number of wcs, n=balls, k=boxes


def resMat(mat):
        mat[1][0] = 1.0-mat[0][0]
        mat[0][1] = 1.0-mat[1][1]
        return mat


def normize(mat):
        for i in xrange(len(mat)):
                mat[:,i] /= sum(mat[:,i])
        return mat


def corrEv(mat, tol=TOL):
        '''
        INPUT:
                mat :: List<List<Float>>
                        # 2x2
        OUTPUT:
                NPArray<Float>
                        # returns eigenvector corresponding to eigenvalue of 1
        '''
        info = np.linalg.eig(mat)
        ev = info[1][:,0] if abs(info[0][0] - 1.0) < tol else info[1][:,1]
        return ev/sum(ev)


def corrEv_m(mat, tol=TOL):
        '''
        INPUT:
                mat :: List<List<Float>>
                        # square
        OUTPUT:
                NPArray<Float>
                        # returns eigenvector corresponding to eigenvalue of 1
        '''
        info = np.linalg.eig(mat)
        for i in xrange(len(info[1][:,0])):
                if abs(info[0][i] - 1.0) < tol:
                        ev = info[1][:,i]
                        return ev/sum(ev)


def devectorize(vm, bins):
        '''
        INPUT:
                vm :: NPArray<Float>
                        # bins^2-length vectorized P^I
                bins :: Integer
                        # the number of bins
        OUTPUT:
                NPArray<NPArray<Float>>
                        # devectorized vm as bins-x-bins-matrix
        '''
        output = np.zeros([bins, bins])
        col = 0
        for i in xrange(0, len(vm), bins):
                output[:,col] = vm[i:i+bins]
                col += 1
        return output


#
# Source: (2nd answer in link)
#  https://stackoverflow.com/questions/4647120/next-composition-of-n-into-k-parts-does-anyone-have-a-
#
def weak_compositions(balls, boxes, parent=tuple()):
        if boxes > 1:
                for i in xrange(balls + 1):
                        for x in weak_compositions(i, boxes - 1, parent + (balls - i,)):
                                yield x
        else:
                yield parent + (balls,)


#
# Source: LeetCode
#
def permute(nums):
        perms = [[]]
        for n in nums:
                new_perms = []
                for perm in perms:
                        for i in range(len(perm)+1):
                                new_perms.append(perm[:i] + [n] + perm[i:])
                perms = new_perms
        return perms
```

```
def perm(n, r):
        return factorial(n)/factorial(n-r)


def genStat(n):
        '''
        INPUT:
                n :: Integer
                        # size of stationary distribution
        OUTPUT:
                List<Float>
                        # generate a random stationary distribution
        '''
        samp = np.random.uniform(0,1,n)
        return samp/sum(samp)


def listMatch(l1, l2, tol=TOL):
        '''
        INPUT:
                l1, l2 :: List<numeric>
                        # lists must be the same length
                tol :: Float
        OUTPUT:
                True if all elements in l1, l2 are equal within tol, else False
        '''
        ll1 = len(l1)
        if ll1 != len(l2):
                print '\n ERROR - listMatch() - Lists have unequal length!\n'
                quit()
        for i in xrange(ll1):
                if abs(l1[i] - l2[i]) >= tol:
                        return False
        return True


def matMatch(m1, m2, tol=TOL):
        '''
        INPUT:
                m1, m2 :: List<List<numeric>>
                        # matrices must have the same shapes
                tol :: Float
        OUTPUT:
                True if all elements in m1, m2 are equal within tol, else False
        '''
        lm1 = np.shape(m1); lm2 = np.shape(m2)
        if lm1[0] != lm2[0] or lm1[1] != lm2[1]:
                print '\n ERROR - matMatch() - Matrices have unequal shapes!\n'
                return False
        for i in xrange(lm1[0]):
                for j in xrange(lm1[1]):
                        if abs(m1[i][j] - m2[i][j]) >= tol:
                                return False
        return True


def stochTest(mat, tol=TOL):
        '''
        INPUT:
                mat :: List<List<Float>>
                        # square matrix
                tol :: Float
        OUTPUT:
                True if columns sum to 1 within tol, else False
        '''
        s = np.shape(mat)
        for i in xrange(len(mat)):
                if abs(sum(mat[:,i]) - 1.0) >= tol:
                        return False
        return True
```

11.12 Graphing Code

```
# Packages
# numpy already imported as np
import sys
import pprint as pp
import matplotlib.pyplot as plt
from time import time
from HELPERS_clean import genStat, normize, listMatch
from TIMEOUT_clean import timeout

# Algorithms
from bS_clean import *
from rS_clean import *
from brS_clean import *
from LP_clean import *
from NRS_clean import *
from GRS_clean import *
from GI_clean import *
from CMAES_clean import *
from VSEA_clean import *
from Series_clean import *

# test cases to generate P^I - (n, pi)
TESTS_PI = [ \
    (2, np.array([.5,.5])), \
    (2, np.array([.25,.75])), \
    (2, np.array([.75,.25])), \
    (2, np.array([.6943,.3057])), \
    (5, np.array([.2,.1,.3,.01,.39])), \
    (10, np.array([.2/2,.1/2,.3/2,.01/2,.39/2,.2/2,.1/2,.3/2,.01/2,.39/2])), \
    (15, np.array([.2/3,.1/3,.3/3,.01/3,.39/3,.2/3,.1/3,.3/3,.01/3,.39/3,.2/3,.1/3,.3/3,.01/3,.39/3])
]

# test cases to generate P^M- (N, n, P^I)
TESTS_PM = [ \
    (2, 2, np.array([[.25, .25],[.75, .75]])), \
    (10, 2, np.array([[.25, .65],[.75, .35]])), \
    (14, 2, np.array([[.25, .65],[.75, .35]])), \
    (21, 2, np.array([[.25, .65],[.75, .35]])), \
    (30, 2, np.array([[.25, .65],[.75, .35]])), \
    (32, 2, np.array([[.25, .65],[.75, .35]])), \

    (2, 3, np.array([[.2, .05, .01],[.7, .35, .9],[.1, .6, .09]])), \
    (2, 4, np.array([[.25, .60, .75, .1],[.1, .1, .05, .2],[.3, .25, .05, .3],[.1, .1, .05, .2]])), \
    (2, 5, np.array([[.3, .65, .05, .2, .4],[.1, .1, .05, .2, .05],[.4, .05, .8, .2, .1],[.1, .1, .05

    (4, 4, np.array([[.25, .60, .75, .1],[.1, .1, .05, .2],[.3, .25, .05, .3],[.1, .1, .05, .2]])), \
    (14, 4, np.array([[.25, .60, .75, .1],[.1, .1, .05, .2],[.3, .25, .05, .3],[.1, .1, .05, .2]]))
]
```

```
##########################################################
def graph(nfuns, domain, rng, labels, xl=None, yl=None, title=None, logx=False, logy=False):
    '''
    INPUT:
        nfuns :: Integer
        domain, rng :: List<Integer>
        labels :: List<String>
    '''
    lines = []
    if not (logx or logy):
        f = plt.plot
    elif logx and not logy:
        f = plt.semilogx
    elif not logx and logy:
        f = plt.semilogy
    elif logx and logy:
        f = plt.loglog
    for i in xrange(nfuns):
        aLine, = f(domain, rng[i], label=labels[i])
        lines.append(aLine)
    plt.legend(handles=lines)
    plt.xlabel(xl)
    plt.ylabel(yl)
    plt.title(title)
    plt.show()

def expRT(n):
    '''
    Expected runtime of GI given n, the number of bins
    '''
    from scipy.special import digamma
    from mpmath import euler
    from scipy.misc import comb
    A = comb(n,2)
    return A*(digamma(A+1.0) + float(euler))

def divide(a,b):
    return float(a)/float(b)

vecexp = np.vectorize(expRT)
vecdiv = np.vectorize(divide)

##########################################################
# Note that the digit-precision is constant between and within all test suites
# system arguments: 2, n2, ur, m
NUM_TIMES = 30 # number of trials = times we run each function (per each epsilon) per each test case
START_CASE = 2 # minimum value of n
if sys.argv[1] == '2':
    print '"n=2"-case direct methods test suite'
    #   vary epsilon
    #   plot log(TIME) x log(n)
```

```
        TEST_F = [bS, rS, brS]
        D = ['bS', 'rS', 'brS']
        NUM_EPSILONS = 10
        TEST_EPSILONS = [1.0/(10.0**i) for i in xrange(1,NUM_EPSILONS+1)]
        NUM_PIS = 4 # number of test cases (test pi values)
        results = [[[0]*NUM_PIS for x in xrange(NUM_EPSILONS)] for y in xrange(len(TEST_F))]
        for f in xrange(len(TEST_F)):
            print '\nFUNCTION:', f
            for T in xrange(len(TEST_EPSILONS)):
                print '\tEPSILON:', T, ', VALUE:', TEST_EPSILONS[T]
                for case in xrange(NUM_PIS):
                    print '\t\tTESTCASE:', case, ', VALUE:', TESTS_PI[case][1]
                    TO_AVG = [] # we will average the values in this array to get the per function per ep
                    for x in xrange(NUM_TIMES): # we'll average time for each test here
                        start = time()
                        print '\t\t\tTRIAL:', x
                        TEST_F[f](2, TESTS_PI[case][1], tol=TEST_EPSILONS[T])
                        TO_AVG.append(time() - start)
                    results[f][T][case] = np.average(TO_AVG)
        print D
        res = np.array(results)
        for f in xrange(len(TEST_F)):
            print '\nRESULTS for FUNCTION', f
            print res[f]
        plot_res = [[0]*NUM_EPSILONS for T in xrange(len(TEST_F))] # each column is an epsilon, each rows
        for f in xrange(len(TEST_F)):
            for T in xrange(NUM_EPSILONS):
                plot_res[f][T] = np.average(res[f,T,:]) # average over all test cases
        for f in xrange(len(TEST_F)):
            print '\nGRAPHING DATA for FUNCTION', f
            print plot_res[f]
        graph(len(TEST_F), TEST_EPSILONS, plot_res, D, \
            xl='Epsilon Value', yl='Runtime (seconds)', title='\"n=2\"-Case Direct Methods', logx=True)
    ################################################
    elif sys.argv[1] == 'n2':
        print '"n>=2"-case direct methods test suite'
        #   vary n (cases from TESTS_PI)
        #   plot f(TIME) x f(n), where f is worst-case complexity of respective method
        TEST_F = [LP, GI, exprRT]
        D = ['LP', 'GI', 'Expected GI']
        MAX_PIS = 7 # maximum value of n+1; a value of 4 will test 2 cases: n=2,3
        results = [[0]*(MAX_PIS-START_CASE) for y in xrange(len(TEST_F)-1)]
        varis = [[0]*(MAX_PIS-START_CASE) for y in xrange(len(TEST_F)-1)]
        for f in xrange(len(TEST_F)-1):
            print '\nFUNCTION:', f
            for case in xrange(START_CASE, MAX_PIS):
                print '\t\tn_VALUE:', case
                TO_AVG = [] # we will average the values in this array to get the per function per epsilo
                for x in xrange(NUM_TIMES): # we'll average time for each test here
                    test_stat = genStat(case)
                    start = time()
                    print '\t\t\tTRIAL:', x, ', pi_VALUE:', test_stat
                    print results
                    r = TEST_F[f](case, test_stat)
                    print 'RESULT:', r
                    print 'Worked?:', listMatch(np.dot(r, test_stat), test_stat)
                    TO_AVG.append(time() - start)
                results[f][case-START_CASE] = np.average(TO_AVG)
                varis[f][case-START_CASE] = np.var(TO_AVG)
        results.append(vecexp(range(START_CASE, MAX_PIS))) # get expected runtime of GI
        varis.append([None])
        print '\n', D
        res = np.array(results)
        v = np.array(varis)
        for f in xrange(len(TEST_F)):
            print '\nRESULTS for FUNCTION', f
            print res[f]
            print 'VARIANCES for FUNCTION', f
            print v[f]
        # only LP
        graph(1, range(START_CASE, MAX_PIS), [res[0]], [D[0]], \
            xl='n Value', yl='Runtime (seconds)', title='\"n>=2\"-Case Reliable Direct Methods')
        # graph GI and its expected runtime, 4 different ways
        graph(2, range(START_CASE, MAX_PIS), res[1:], D[1:], \
            xl='n Value', yl='Runtime (seconds)', title='\"n>=2\"-Case Reliable Direct Methods', logx=Tru
        graph(2, range(START_CASE, MAX_PIS), res[1:], D[1:], \
            xl='n Value', yl='Runtime (seconds)', title='\"n>=2\"-Case Reliable Direct Methods', logx=Tru
        graph(2, range(START_CASE, MAX_PIS), res[1:], D[1:], \
            xl='n Value', yl='Runtime (seconds)', title='\"n>=2\"-Case Reliable Direct Methods', logx=Fal
        graph(2, range(START_CASE, MAX_PIS), res[1:], D[1:], \
            xl='n Value', yl='Runtime (seconds)', title='\"n>=2\"-Case Reliable Direct Methods')
        # see difference between expected GI and actual GI real-time runtimes
        print '\nRESULTS for GI/E[GI]'
        print vecdiv(results[1],results[2])
        graph(1, range(START_CASE, MAX_PIS), [vecdiv(results[1],results[2])], ['Ratio GI/E[GI]'], \
            xl='n Value', yl='Ratio (GI)/(Expected GI) Realtime Runtime', title='\"n>=2\"-Case Direct Met
        # graph everything
        graph(2, range(START_CASE, MAX_PIS), res, D, \
            xl='n Value', yl='Runtime (seconds)', title='\"n>=2\"-Case Direct Methods', logx=False, logy=
    ################################################
    elif sys.argv[1] == 'ur':
        print 'un-reliable (NRS, GRS, CMAES) direct methods test suite'
        #   vary n (cases randomly generated from genStat())
        #   plot proportion_of_correct_answers_within_an_hour x test_case_index
        MAX_SECS = 60
        TEST_F = [NRS, GRS, CMAES]
        D = ['NRS', 'GRS', 'CMA-ES']
        MAX_PIS = 6 # maximum value of n+1; a value of 4 will test 2 cases: n=2,3
        results = [[0]*(MAX_PIS-START_CASE) for y in xrange(len(TEST_F))]
        for f in xrange(len(TEST_F)):
            print '\nFUNCTION:', f
            for case in xrange(START_CASE, MAX_PIS):
                test_stat = genStat(case)
                print '\tn_VALUE:', case, ', pi_VALUE:', test_stat
                TO_AVG = [] # we will average the values in this array to get the per function per epsilo
                for x in xrange(NUM_TIMES): # we'll average time for each test here
                    print '\t\tTRIAL:', x
                    TO_AVG.append(timeout(MAX_SECS*case, TEST_F[f], case, test_stat))
                results[f][case-START_CASE] = np.average(TO_AVG)
        print D
        res = np.array(results)
        for f in xrange(len(TEST_F)):
            print '\nRESULTS for FUNCTION', f
            print res[f]
        graph(len(TEST_F), range(START_CASE, MAX_PIS), res, D, \
            xl='n Value', yl='Proportion of '+str(NUM_TIMES)+' Successful Trials Completed in an Hour', t
    ################################################
    elif sys.argv[1] == 'm':
        print 'P^(M) generating methods test suite'
        #   vary n and N (cases form TESTS_PM)
        #   plot TIME x test_case_index
        TEST_F = [VSEA, Series]
        D = ['VSEA', 'Series']
        NUM_TESTS = 10 # the number of tests to run: 0 to len(TESTS_PM)
        NUM_TESTS = min(max(NUM_TESTS,0), len(TESTS_PM)) # quality control

        results = [[0]*NUM_TESTS for y in xrange(len(TEST_F))]
        for f in xrange(len(TEST_F)):
            print '\nresults', results
            print '\nFUNCTION:', f
            for case in xrange(NUM_TESTS):
                print '\tTESTCASE:', case, ', CASE_VALUE:\n\t', TESTS_PM[case]
                start = time()
                TEST_F[f](TESTS_PM[case][0], TESTS_PM[case][1], TESTS_PM[case][2])
                results[f][case] = time() - start
        print D
        res = np.array(results)
        for f in xrange(len(TEST_F)):
            print '\nRESULTS for FUNCTION', f
            print res[f]
        graph(1,len(TEST_F), range(NUM_TESTS), res, D, \
            xl='Test Case', yl='Runtime (seconds)', title='Mass Matrix Generation Methods')
        graph(1, range(NUM_TESTS), [res[0]], [D[0]], \
            xl='Test Case', yl='Runtime (seconds)', title='Mass Matrix Generation Methods - VSEA')
        graph(1, range(NUM_TESTS), [res[1]], [D[1]], \
            xl='Test Case', yl='Runtime (seconds)', title='Mass Matrix Generation Methods - Series')
```

11.13 Timeout Helper Function

```
# Source:
#   https://stackoverflow.com/questions/492519/timeout-on-a-function-call

def timeout_inny(secs, fun, *params):
    '''
    Function 'fun' with arguments 'params' ~may~ run for 'secs' seconds.
    '''
    import signal

    # Register an handler for the timeout
    def handler(signum, frame):
        raise Exception("FROM 'timeout(): fun has runout of 'secs' time!")

    signal.signal(signal.SIGALRM, handler) # Register the signal function handler
    signal.alarm(secs) # Define a timeout for function

    # Run function until timeout
    try:
        fun(*params)
        return 1
    except Exception, exc:
        print exc
        return 0


def timeout(secs, fun, *params):
    '''
    Function 'fun' with arguments 'params' ~may~ run for 'secs' seconds.
    '''
    import signal
    if timeout_inny(secs, fun, *params) == 1:
        signal.alarm(0) # end signal thread!
        return 1
    else:
        return 0

# FOR TESTING:
# from time import sleep
# from TIMEOUT_clean import timeout
# def t(n):
#     sleep(n)
#     return 0
# timeout(5,t,2)
# timeout(2,t,5)
```

12 Bibliography

References

[Hua77] Cheng-Chi Huang. "Non-homogeneous Markov chains and their applications". PhD thesis. Ames, IA, 1977.

[GG84] S. Geman and D. Geman. "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6.6 (Nov. 1984), pp. 721–741. ISSN: 0162-8828. DOI: `10.1109/TPAMI.1984.4767596`.

[GZ93] Oded Galor and Joseph Zeira. "Income Distribution and Macroeconomics". In: *The Review of Economic Studies* 60.1 (1993), pp. 35–52.

[HK03] Taher H. Haveliwala and Sepandar D. Kamvar. "The Second Eigenvalue of the Google Matrix". In: *Stanford University Technical Report*. 2003.

[ST08] Daniel A. Spielman and Shang-Hua Teng. "Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time". In: *(pre-print)* (2008).

[LBE11] Mohamed Esseghir Lalami, Vincent Boyer, and Didier El-Baz. "Efficient Implementation of the Simplex Method on a CPU-GPU System". In: *2011 IEEE International Parallel Distributed Processing Symposium* (2011), pp. 1998–2001.

[Pag12] Daniel R. Page. "Generalized Algorithm for Restricted Weak Composition Generation". In: *Journal of Mathematical Modelling and Algorithms* 12.4 (2012), pp. 345–372.

[AG] Quamrul Ashraf and Oded Galor. *The "Out of Africa" Hypothesis, Human Genetic Diversity, and Comparative Economic Development*. Available at `https://assets.aeaweb.org/assets/production/articles-attachments/aer/data/feb2013/20100971_app.pdf` (2018/04/10).

[Csi] Péter Csikvári. *composition$_p$artition.pdf*. Available at `http://math.mit.edu/~csikvari/composition_partition.pdf` (2018/04/10).

[Gal] Oded Galor. *Genetic Diversity and Comparative Development*. Available at `http://media.virbcdn.com/files/8a/78fedbd35de3e73a-Galor-Lecture4-2016-H.pdf` (2018/04/10).

[Hana] Nikolaus Hansen. *gecco2013-CMA-ES-tutorial.pdf*. Available at `https://www.lri.fr/~hansen/gecco2013-CMA-ES-tutorial.pdf` (2018/04/10).

[Hanb] Nikolaus Hansen. *html-pythoncma*. Available at `https://www.lri.fr/~hansen/html-pythoncma/` (2018/04/10).

[Hanc] Nikolaus Hansen. *nameIndex*. Available at `http://cma.gforge.inria.fr/apidocs-pycma/nameIndex.html#R` (2018/04/10).

[Hand] Nikolaus Hansen. *The CMA Evolution Strategy: A Tutorial*. Available at `https://arxiv.org/pdf/1604.00772.pdf` (2018/04/10).

[Kal] Sagar Kale. *Eigenvalues and Mixing Time*. Available at `https://math.dartmouth.edu/~pw/math100w13/kale.pdf` (2018/04/10).

[KS] Jonathan A. Kelner and Daniel A. Spielman. "A Randomized Polynomial-Time Simplex Algorithm for Linear Programming". In: *(pre-print)* ().

[uI] user335262 and Ian. *Example of a markov chain that has a distribution that converges to some limit*. Available at `https://math.stackexchange.com/questions/1762017/example-of-a-markov-chain-that-has-a-distribution-that-converges-to-some-limit` (2018/04/10).

[Wol] WolframAlpha. *Output*. Available at `http://www.wolframalpha.com/input/?i=sum+A%2F(A-j)+from+j%3D0+to+j%3DA-1` (2018/04/10).

[WN] WolframMathWorld and James Noyes. *Linear Programming*. Available at `http://mathworld.wolfram.com/LinearProgramming.html` (2018/04/10).