

PROBABILISTIC INTERPRETATIONS OF VARIABLES IN TRANSIENT M/G/1 RANDOM WALKS

DREW BOLLINGER AND DAVID WHEELER

ABSTRACT. We consider the transient case of M/G/1 random walks that are continuous to the left on the integer points in a half strip, $\mathcal{S} = \{1, 2, \dots\} \times \mathbb{Z}_+$. The long-term position of our walk can be determined through use of a generating function. Although in the transient case the long-term probability of being at any point is zero, the matrices G_{min} and G_{max} , which are used in solving the generating function, still yield many variables with probabilistic significance. This paper explores probabilistic interpretations of the variables in the equation $L \cdot G_{min} = \rho \cdot L$

1. INTRODUCTION

We consider the transient case of M/G/1 random walks on the integer points in a half strip,

$$\mathcal{S} = \{1, 2, \dots, M\} \times \{0, 1, 2, \dots\} = \{1, 2, \dots\} \times \mathbb{Z}_+,$$

where \mathbb{Z}_+ denotes the nonnegative integers. Our attention is further restricted to homogeneous random walks that are continuous to the left. Note that when referencing our position throughout the paper, we will colloquially refer to our first component as *phase* and our second as *level*. To interpret the quantities we are studying, it is convenient to introduce a special *stopping time*. Remember that $X_t = (S_t, Q_t) \in \mathcal{S}$ is the location in the strip \mathcal{S} of the random walk at time $t = 0, 1, 2, \dots$. Let τ denote the first time t for which X_t is at level 0; i.e.

$$\tau = \min\{t : Q_t = 0\},$$

where $\tau = +\infty$ if the random walk is never at level 0; i.e. $Q_t > 0$ for all $t = 0, 1, 2, \dots$. The random variable τ is a stopping time because one can determine if $\tau \leq k$ by examining the values of X_0, X_1, \dots, X_k . If $\tau \wedge t = \min\{t, \tau\}$, $t = 0, 1, 2, \dots$, then the sequence of random variables $Y_t = X_{\tau \wedge t}$, $t = 0, 1, 2, \dots$ is the random walk that results if the *boundary states*, i.e. the states at level 0, are made absorbing. Thus, such quantities as “the probability that one is at phase j on the first entrance to level 0” are equal to $P\{X_\tau = (j, 0)\}$.

Date: November 19, 2006.

The research of the authors was supported by generous grants from the University of Michigan provided by the National Science Foundation. We wish to extend our gratitude to our research advisor, B.A. Taylor, for his invaluable assistance.

2. NOTATION AND PRELIMINARIES

Definition 1. Let $a(z)$ be the distribution function of the steps in the random walk taken from level one or higher. That is,

$$a(z) = a_0 + a_1z + \cdots + a_mz^m = \sum_{k=0}^m a_k z^k$$

where a_k is an $M \times M$ matrix whose i, j -th entry is the probability that we move $k-1$ levels to the right into phase j , given that we began in phase i . We also require that $a(1)$ be irreducible and $a(1) \cdot \bar{1} = \bar{1}$, where $\bar{1}$ is a $1 \times M$ vector with 1 as every entry, to ensure that $a(1)$ is a stochastic matrix.

Definition 2. Let $b(z)$ be the distribution function of the steps in the random walk at the boundary (level 0). That is,

$$b(z) = b_0 + b_1z + \cdots + b_tz^t = \sum_{k=0}^t b_k z^k$$

where b_k is an $M \times M$ matrix whose i, j -th entry is the probability that we move k levels to the right into phase j , given that we began in phase i . We similarly require that $b(1) \cdot \bar{1} = \bar{1}$ to ensure that $b(1)$ is a stochastic matrix.

Let $X_t \in \mathcal{S}$, $t = 0, 1, \dots$, denote the position of the random walks at discrete times $t = 0, 1, \dots$. The matrix entry, $(a_k)_{i,j}$, determines the transition probabilities for one step in the random walk, from $X_k = (i, m)$, $m \geq 1$, to $X_{k+1} = (j, n)$, in the following way:

$$P\{X_{k+1} = (j, n) \mid X_k = (i, m)\} = (a_{n-m+1})_{i,j}$$

Definition 3. Let $\varphi_k = [\pi_{k,1} \cdots \pi_{k,i} \cdots \pi_{k,n}]$, where $\pi_{k,i}$ is the probability of being in level k , phase i at time t as $t \rightarrow \infty$. Further,

$$\varphi(z) = \varphi_1 + \varphi_2z + \cdots = \sum_{k=1}^{\infty} \varphi_k z^{k-1}$$

This $1 \times M$ vector-valued function, $\varphi(z)$, must satisfy the generating function equation:

$$\varphi(z)(z - a(z)) + \varphi_0(I - b(z)) = 0 \tag{1}$$

It is natural to normalize any non-trivial solution so that the sum of the probabilities is 1. Thus, we require that

$$(\varphi_0 + \varphi(1)) \cdot \bar{1} = 1$$

It is known that a non-trivial solution, $\varphi(z)$ exists if and only if the expected step in the random walks is to the left.

To solve (1), we use an $M \times M$ matrix, G , which satisfies the condition:

$$G = a(G) \tag{2}$$

In the left-drifting case, this G matrix is unique, whereas in the transient case, there are at least two solutions. One will be sub-stochastic, denoted G_{min} , and it is the minimal nonnegative solution of the equation (2). There is also at least one stochastic solution which we will denote by G_{max} . It is possible to construct G_{max} ,

the stochastic matrix which satisfies (2), using information from G_{min} . First, we find the largest eigenvalue, ρ , of G_{min} and solve for L in the following equations:

$$L \cdot G_{min} = \rho \cdot L \quad (3)$$

$$L \cdot \bar{1} = 1 \quad (4)$$

Then we construct G_{max} in the following way:

$$G_{max} = G_{min} + (I - G_{min}) \cdot \bar{1} \cdot L \quad (5)$$

where L is the vector solved for in (3) and (4).

3. PROBABILISTIC INTERPRETATIONS

We first give the known probabilistic interpretation of the sub-stochastic matrix, G_{min} :

Theorem 1. *Let G_{min} be the minimal, non-negative $M \times M$ sub-stochastic matrix which satisfies (2). The i, j -th entry of G_{min} is equal to*

$$P\{X_\tau = (j, 0) \mid X_0 = (i, 1)\}$$

From this, we seek to determine the probabilistic interpretations of the other variables in (3). To begin, we determine the probabilistic interpretations of vector and matrix multiplication:

Lemma 1. *Let $V = (v_1, \dots, v_M)$ where $v_i = P\{X_0 = (i, 1)\}$, and suppose that $V \cdot \bar{1} = 1$. Then the j -th component of $V \cdot G_{min}$ is equal to*

$$\sum_i P\{X_\tau = (j, 0) \mid X_0 = (i, 1)\} \cdot v_i.$$

Proof.

$$\begin{aligned} & \sum_i P\{X_\tau = (j, 0) \mid X_0 = (i, 1)\} \cdot v_i \\ &= \sum_i v_i \cdot (G_{min})_{i,j} \\ &= (v \cdot G_{min})_j \end{aligned}$$

□

Lemma 2.

$$(G_{min} \cdot \bar{1})_i = P\{\tau < \infty \mid X_0 = (i, 1)\}$$

Proof.

$$\begin{aligned} & P\{\tau < \infty \mid X_0 = (i, 1)\} \\ &= \sum_j P\{X_\tau = (j, 0) \mid X_0 = (i, 1)\} \\ &= \sum_j (G_{min})_{i,j} \\ &= G_{min} \cdot \bar{1} \end{aligned}$$

□

Proposition 1. $L = \{\ell_1 \dots \ell_j \dots \ell_n\}$ is the probability given by:

$$\ell_j = \lim_{N \rightarrow \infty} P\{X_\tau = (j, 0) \mid \tau < \infty, X_0 = (i, N)\}$$

Proof. From Theorem 1 we know that The i, j -th entry of G_{min} represents:

$$P\{X_\tau = (j, 0) \mid X_0 = (i, 1)\}$$

The N -th iteration of this Markov Chain is represented by $(G_{min})^N$. Let the i, j -th entry of H_{min} be

$$\left(\frac{(G_{min})_{i,j}}{\sum_j (G_{min})_{i,j}} \right)$$

Since we know that G_{min} has right eigenvector L , a computation based upon the power method will show that:

$$\lim_{N \rightarrow \infty} (H_{min})^N = \bar{1} \cdot L$$

$$\ell_j = \lim_{N \rightarrow \infty} \left(\frac{(G_{min})_{i,j}}{\sum_j (G_{min})_{i,j}} \right)^N, \forall i, 1 \leq i \leq M \quad (6)$$

Since $(G_{min})_{i,j}^N$ represents the probability that we will hit zero in phase j from level N , phase i , (6) represents the probability that we will hit zero in phase j from level N phase i given that we have indeed hit zero. Since each row of (6) is equal to L , it follows that ℓ_j is the probability that we hit zero in phase j given that we hit zero and started in any phase in level N . \square

This leads us to a clearer idea of the true meaning of the eigenvalue. ρ .

Proposition 2.

$$\rho = \sum_i P\{\tau < \infty \mid X_0 = (i, 1)\} \cdot \ell_i$$

Proof. From (4) we know that:

$$L \cdot \bar{1} = 1$$

Thus, we can transform (3) as follows:

$$\begin{aligned} L \cdot G_{min} &= \rho \cdot L \\ L \cdot G_{min} \cdot \bar{1} &= \rho \cdot L \cdot \bar{1} \\ L \cdot G_{min} \cdot \bar{1} &= \rho \cdot 1 \\ L \cdot G_{min} \cdot \bar{1} &= \rho \end{aligned} \quad (7)$$

By Lemma 1:

$$L \cdot G_{min} \cdot \bar{1} = \left(\sum_i P\{X_\tau = (j, 0) \mid X_0 = (i, 1)\} \cdot \ell_i \right) \cdot \bar{1}$$

By multiplying out the right side of that equation, we are summing the probabilities $\forall j, 0 \leq j \leq M$. Therefore:

$$\rho = \sum_i P\{\tau < \infty \mid X_0 = (i, 1)\} \cdot \ell_i$$

\square

4. SIMULATIONS AND DATA

We first used simulations to verify the known probabilistic interpretation of G_{min} . By creating a program to produce "dots" and then move them according to (1), we can examine the probabilities of different events. In this particular simulation (source code can be found in Appendix A) we sought to find the probability that the "dot" eventually hit zero in the j -th phase, given that it began at level 1, phase i . The table below shows typical error on an example system:

		Steps			
		100	250	500	1000
Runs	100	.3542	.3551	.3567	.2141
	1000	.0949	.1155	.1094	.0649
	10000	.0402	.0496	.0352	.0339
	100000	.0125	.0133	.0058	.0067

We can see that as the runs and steps (number of steps we allowed the "dot" to travel before breaking) increased, our error becomes very close to zero. The ideal system would have infinite steps to allow the "dot" as much time as necessary to hit zero, however, our computing power did not allow for this. We computed error by constructing a matrix of how many times a "dot" hit level zero in phase j given that it began in level 1, phase i . We then normalized the matrix, subtracted it from G_{min} and summed the absolute value of every entry.

We also hoped to confirm the formula used to calculate the expected step in the random walk of a particular $a(z)$ function. The formula used to calculate the expected step is as follows:

$$\pi \cdot (a'(1) \cdot \bar{1} - a(1) \cdot \bar{1})$$

where π satisfies the equations:

$$\pi \cdot a(1) = \pi$$

$$\pi \cdot \bar{1} = 1$$

We then compared this result to the average steps a "dot" had moved over an arbitrary time.

	Runs		
	100	1000	10000
Simulated Velocity	.3209	.3213	.3224
Calculated Velocity	.3229	.3229	.3229
Error	0.62%	0.50%	0.15%

Here we computed error by the standard method of

$$\frac{|V_C - V_S|}{V_C}$$

where V_C and V_S represent the calculated and simulated velocities respectively.

Next we hoped to further confirm our propositions with simulation data. We first prepared a simulation to verify Proposition 1. We ran four trials with varying initial levels (2-5). As our initial level went to infinity, our error should decrease. To guarantee equally accurate results for each trial, we increased the number of runs as we moved further from zero to ensure similar amounts of "dots" would return to zero.

	Initial Level			
	2	3	4	5
Error	.1362	.0468	.0369	.0147

We computed the error of this process in a similar way to our G_{min} computations. We again constructed a matrix of how many times a "dot" hit level zero in phase j given that it began in level 1, phase i and proceeded to normalize it. Then we subtracted it from $\bar{1} \cdot L$ and summed the absolute values of the entries (We use $\bar{1} \cdot L$ because we assume that as we approach infinity, the rows will be equal, independent of starting phase). Also, since we are simulating movement back to level zero, we note that a normalized $(G_{min})^N$, where N is our initial level, helps us to calculate our expected error at each level. We merely have to substitute the normalized $(G_{min})^N$ for our simulation matrix in the example calculations below:

$$\begin{aligned} \bar{1}^T \cdot \left| (\bar{1} \cdot L) - \left(\frac{(G_{min})_{i,j}}{\sum_j (G_{min})_{i,j}} \right)^2 \right| \cdot \bar{1} &= .1183 \\ \bar{1}^T \cdot \left| (\bar{1} \cdot L) - \left(\frac{(G_{min})_{i,j}}{\sum_j (G_{min})_{i,j}} \right)^3 \right| \cdot \bar{1} &= .0247 \\ \bar{1}^T \cdot \left| (\bar{1} \cdot L) - \left(\frac{(G_{min})_{i,j}}{\sum_j (G_{min})_{i,j}} \right)^4 \right| \cdot \bar{1} &= .0053 \\ \bar{1}^T \cdot \left| (\bar{1} \cdot L) - \left(\frac{(G_{min})_{i,j}}{\sum_j (G_{min})_{i,j}} \right)^5 \right| \cdot \bar{1} &= .0011 \end{aligned}$$

From (6) we know that as $N \rightarrow \infty$, the error will converge to 0.

To run a simulation to compute ρ , we made sure to begin each dot in level 1, phase i with probability ℓ_i . After we knew that our starting phase would be properly distributed, we just had to count how many times our "dot" hit zero compared to the total number of runs. The table below shows typical error on an example system:

		Steps			
		100	250	500	1000
Runs	100	0.0587	0.0287	0.0514	0.0314
	1000	0.0027	0.0094	0.0234	0.0074
	10000	0.0025	0.0030	0.0056	0.0008
	100000	0.0004	0.0019	0.0005	0.0001

When given enough time (steps) and many runs, we see that the error

$$\left| \rho - \frac{\text{total hits at zero}}{\text{total runs}} \right|$$

approaches zero.

5. ALGORITHMS

5.1. Random Walk Simulation. Having an accurate simulation of the random walk process is crucial to ensure the validity of our experimental observations. The simulation keeps track of the coordinates of a point and moves the point according to the system's distribution rules. The process is as follows:

- Step 1:** At the beginning of each step in the process, the simulation chooses the appropriate distribution rule, depending on the position of the point: $b(z)$ is used if the point is on the boundary; $a(z)$ is used otherwise.
- Step 2:** The simulation then partitions the range $[0,1]$ in proportion to the entries in the i -th row of each a_k , where i is the starting phase of the point. Figure 1 demonstrates this partitioning.
- Step 3:** Next, the simulation chooses a random number in the range $[0,1]$, and selects an entry based on that number.
- Step 4:** The simulation then moves the dot according to the chosen entry. For example, in Figure 1, the random number falls in the range of $(a_1)_{i,3}$. This means the dot stays in the same level, but moves to phase 3.

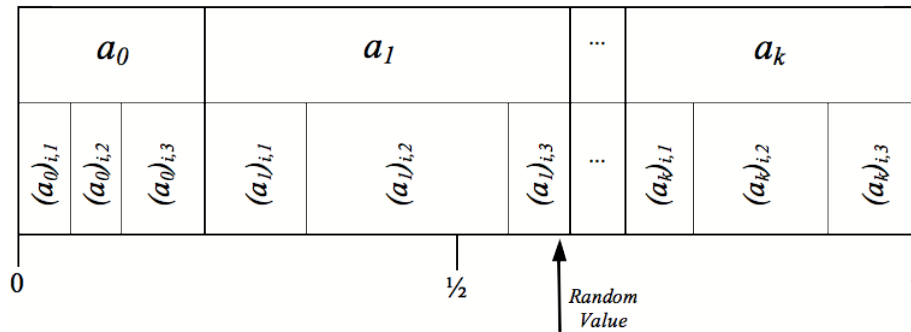


FIGURE 1. Random Walk Simulation

5.2. Bollinger-Wheeler Reduced Row Echelon Form. This is an algorithm that returns the reduced row echelon form of a matrix while correcting for small rounding errors inherent in floating point operations. The process is much the same as the standard row reduction algorithm, but it zeroes out values that are arbitrarily close to zero. The pseudo-code below describes the process:

Pseudo-code notation:

Diamonds (\diamond) represent loops, and the commands underneath them should be executed as many times as necessary to satisfy the condition given on these lines.

Bullets (\bullet) represent single commands, and should only be executed once per loop.

Forward Substitution:

\diamond For each column j of the matrix R , starting from the left:

- Find the row i whose entry in column j has the largest absolute value.
- Switch rows i and j .
- Divide every entry in row j by the $R_{j,j}$.
- ◇ For each row r below row j .
 - Add $-R_{r,j}$ times row j to row r .
 - ◇ For each nonzero entry in row r :
 - If the absolute value of the entry is below some threshold value, then let that entry equal zero.

Back Substitution:

- ◇ For each column j of the matrix R , starting from the right:
 - Find the bottom-most nonzero entry i in column j .
 - ◇ For each row r above row i :
 - Add $-\frac{R_{r,j}}{R_{i,j}}$ times row j to row r .
 - ◇ For each nonzero entry in row r :
 - If the absolute value of the entry is below some threshold value, then let that entry equal zero.

5.3. “Wheeler Whirlwind” Rule Generation Algorithm. Due to the large number of variables present in the distribution rules $a(z)$ and $b(z)$, we required a method to pick a rule at random from the entire space, while still ensuring that $a(1)$ would be stochastic. The pseudo-code below describes the process of generating a rule with N phases and matrices $a_0 \dots a_k$:

- ◇ For each phase i :
 - Let $total = 1$.
 - ◇ Pick a random integer p in the range $[1,k]$, and repeat until k integers have been chosen.
 - Pick a random floating-point number r in the range $[0,total]$.
If p is the last possible selection, then let $r = total$.
 - Subtract r from $total$.
 - Let $ration = r$
 - ◇ Pick a random integer q in the range $[1,N]$, and repeat until N integers have been chosen.
 - Pick a random floating-point number s in the range $[0,ration]$.
If s is the last possible selection, then let $s = ration$.
 - Subtract s from $ration$
 - Let $(a_p)_{i,q} = s$

Below are two examples of rules generated by this algorithm, and graphical representations of the probability of movement:

$$p(z) = \begin{bmatrix} .0288 & .1842 \\ .1602 & .1057 \end{bmatrix} + \begin{bmatrix} .4220 & .0099 \\ .2283 & .0295 \end{bmatrix} z + \begin{bmatrix} .0639 & .2912 \\ .2389 & .2374 \end{bmatrix} z^2$$

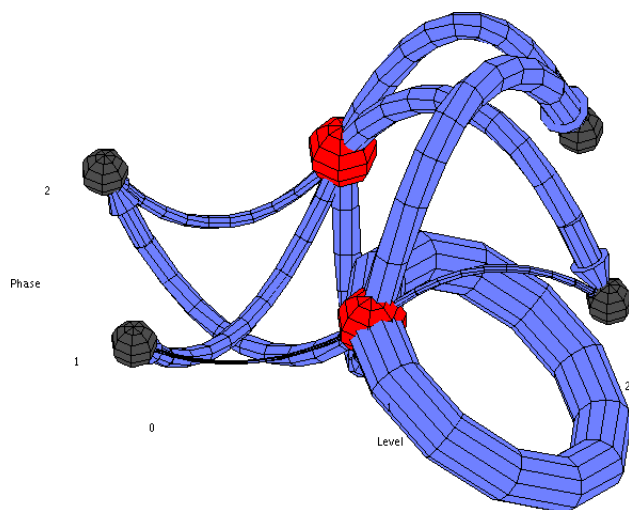


FIGURE 2. Graphical Representation of $p(z)$

$$q(z) = \begin{bmatrix} .0869 & .0491 & .0021 \\ .1775 & .0157 & .2985 \\ .0020 & .0105 & .0054 \end{bmatrix} + \begin{bmatrix} .0064 & .0036 & .0100 \\ .0016 & .0001 & .0000 \\ .0267 & .0797 & .0067 \end{bmatrix} z + \begin{bmatrix} .0184 & .0104 & .0653 \\ .0002 & .0015 & .0000 \\ .0002 & .0006 & .0013 \end{bmatrix} z^2 + \begin{bmatrix} .3583 & .0359 & .3538 \\ .0920 & .0367 & .3765 \\ .0615 & .4487 & .3568 \end{bmatrix} z^3$$

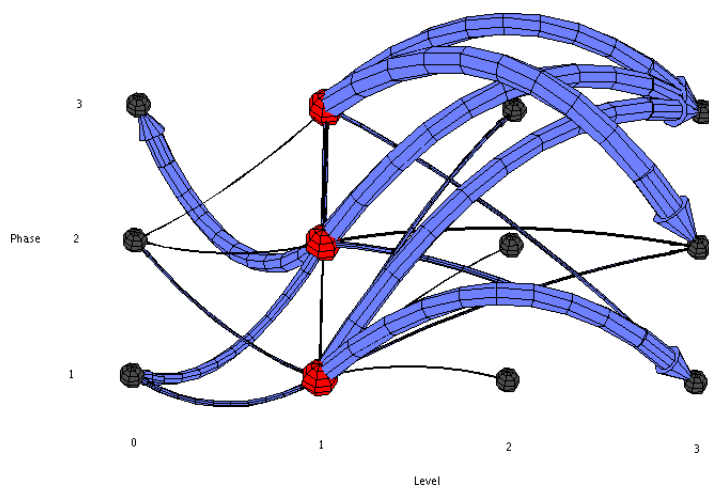


FIGURE 3. Graphical Representation of $q(z)$

6. ONWARD HO!

An important variable whose probabilistic interpretation was not discussed yet in this paper is G_{max} . One of our initial goals was to discover such an interpretation, but alas, we did not find one. We tried to discern its meaning based upon information found within equation (5). We noted that G_{max} was made up of G_{min} and the *deficiency*, $(I - G_{min})$, multiplied by $\bar{1}$ and L . We found this construction rather odd because it combines the runs which never return to zero ($\tau = \infty$) and the phase distribution on runs which do return to zero, L . It seemed to us that this was an artificial way of constructing a stochastic matrix which has a stabilizing vector of L , i.e.

$$L \cdot G_{max} = L$$

This matrix warrants further study to determine if it has any significant probabilistic interpretation.

7. APPENDIX A: SOURCE CODE

Note: All code was written in Maple 10.

```
#####
##
##
##
##          RANDOM WALK TOOLS
##
##          Copyright 2006 David Wheeler and Drew Bollinger
##
##
##
#####
##
##
## OVERVIEW:
##   This file defines the RandomWalk package, which contains the tools and
##   functions used by our simulations, as well as functions to calculate
##   useful variables for arbitrary M/G/1 Random Walks.
##
##
## USAGE NOTES:
##   Typical usage of this software should be as follows:
##   1. Be sure the RandomWalk package is either in your Maple
##      repository, or in a separate repository in the Maple search path.
##   2. Call "with(RandomWalk);" to load the functions in this package.
##   3. Define the variables MatrixListA and (optionally) MatrixListB,
##      either manually or by calling a function in the "Random Walk
##      Generating Functions" section.
##   4. Call "Housekeeping();" to define the rest of the important
##      global variables.
##   5. Call one or more functions from the "Variable Calculation
##      Functions" section.
##
##
## GLOBAL VARIABLES:
##   The below functions depend heavily on global variables defined by
##   this package. These variables must be set before any functions in
##   the "Variable Calculation Functions" section may be called.
##   The user may define The important global variables are:
##
## * MatrixListA - This is a variable of type "list" which contains the
##   transition probability matrices for level 1 and higher. That is,
##   this is a list of every matrix a_k used in the distribution
##   function of the steps in the random walk. Note that MatrixListA[1]
##   represents a_0. There must be at least three matrices in this list,
##   and the matrices must be square and have equal dimensions.
##   In other words, MatrixListA[k][i,j] is the probability that in the
##   next iteration of the process, a point will move (k-1) units to
##   the right from phase i to phase j.
##
##
```

```

## * MatrixListB - This is a list of matrices which represent the transition
## probabilities for level 0 only (see MatrixListA). These matrices must be
## square and have dimensions equal to the matrices in MatrixListA.
##
## * phases - This variable, is the number of phases in the system. This
## variable is called 'M' in our paper.
##
## * eye - The identity matrix with dimensions (phases x phases)
##
## * oneBar - This is a (phases x 1) vector in which every element is 1.
##
## * oneHat - This is a (1 x phases) vector in which every element is 1.
##
## * a(z) - The distribution function of the steps in the random walk
## taken from level 1 or higher. This can be constructed automatically
## from MatrixListA by calling "Housekeeping();" or
## "a:=ProbGenerator(MatrixListA);"
##
## * b(z) - The distribution function of the steps in the random walk
## taken from level 0. This can be constructed automatically
## from MatrixListA by calling "Housekeeping();" or
## "b:=ProbGenerator(MatrixListB);"
##
## * TGE(z) - The generating function equation for the long term
## long term position probabilities for the random walk.
##
#####

RandomWalk := module()
  export ColumnVectorSum, RowVectorSum, RoundAwesome, StochasticMatrix,
    MakeStochastic, ProbList, ProbGenerator, GFuncGenerator, RandomRandomWalk,
    Housekeeping, CalcG, DSubForward, DSubBackward, BollingerWheelerRREF,
    WeightedMatrixList, WeightedRandomWalk, CalcGSubStochastic,
    WheelerWhirlwind, WhirlwindRandomWalk, CalcPhaseDistribution, CalcDrift,
    BiggestEval, FindL, FindPiZero;
  option package;

randomize();

```

```

#####
#####
#####              Miscellaneous Functions          #####
#####              #####
#####
#####

### ColumnVectorSum
#
# Purpose: Returns the sum of the entries in a column vector.
#
ColumnVectorSum:=vec->Multiply(Vector[row](1..Dimension(vec),1),vec);

### RowVectorSum
#
# Purpose: Returns the sum of the entries in a row vector.
#
RowVectorSum:=vec->Multiply(vec,Vector[column](1..Dimension(vec),1));

### RoundAwesome
#
# Purpose: Rounds all the entries in matrix to an arbitrary number of digits
#         and returns the resulting matrix.
#
# NOTE: This function modifies its input!
#
RoundAwesome:=proc(whatevs:Matrix, dij) local y,x;
  for y to RowDimension(whatevs) do
    for x to ColumnDimension(whatevs) do
      #whatevs[y,x]:=evalf[dij](trunc(whatevs[y,x]*10^dij)/(10^dij));
      whatevs[y,x]:=parse(sprintf(cat(cat("%.",dij),"f"),whatevs[y,x]));
    end do;
  end do;
  whatevs;
end proc;

### StochasticMatrix
#
# Purpose: Returns a randomly-generated stochastic matrix.
#
StochasticMatrix:=proc(y,x) local i,coef,ret;
  ret:=RandomMatrix(y,x, generator=1..99.);
  for i from 1 to y
  do
    coef:=1/RowVectorSum(Row(ret,i));
    RowOperation(ret,i,coef,inplace=true);
  end do;
  ret;
end proc;

```

```

### MakeStochastic
#
# Purpose: Divides each row of an input matrix by its respective row sum
#           and returns the resulting matrix.
#
# NOTE: This function modifies its input!
#
MakeStochastic:=proc(mat) local i,coef;
  for i from 1 to ColumnDimension(mat)
  do
    coef:=1/RowVectorSum(Row(mat,i));
    RowOperation(mat,i,coef,inplace=true);
  end do;
  RoundAwesome(mat,30);
  mat;
end proc:

### BiggestEval
#
# Purpose: Returns the largest real eigenvalue of a matrix.
#
BiggestEval:=proc(mat) local x,eVals,rho,test;
  eVals:=Eigenvalues(mat);
  rho:=Re(eVals[1]);
  for x from 2 to Dimension(eVals) do
    test:=Re(eVals[x]);
    if test>rho then rho:=test; end if;
  end do;
  rho;
end proc:

### ProbList
#
# Purpose: Generates a list of (levelSpan) matrices with dimensions
#           (phases x phases), with the additional property that
#           the sum of all the matrices is a stochastic matrix.
#
ProbList:=proc(levelSpan,phases)::function; local x,cols, mat;
  cols:=phases*levelSpan;
  mat:=StochasticMatrix(phases,cols);
  [seq(mat[1..phases,x*phases+1..(x+1)*phases], x=0..floor(cols/phases)-1)];
end proc:

### ProbGenerator
#
# Purpose: Returns a probability distribution function for a list
#           of matrices, usually MatrixListA or MatrixListB.
#
# NOTE: This function accepts only scalar inputs.
#
ProbGenerator:=proc(list)::function; local x;
  z->add(list[x+1]*z^x,x=0..nops(list)-1);
end proc:

```

```

### GFuncGenerator
#
# Purpose: Returns a probability distribution function for a list
#           of matrices, usually MatrixListA or MatrixListB.
#
# NOTE: This function accepts only Matrix inputs.
#
GFuncGenerator:=proc(list)::function; local x;
  z->add(list[x+1].evalf(z)^x,x=0..nops(list)-1);
end proc:

### Housekeeping
#
# Purpose: Defines important global variables.
#           This should be called every time MatrixListA or
#           MatrixListB are edited.
#
Housekeeping:=proc() global a,b,eye, phases, oneBar, oneHat;
  phases:=RowDimension(MatrixListA[1]);
  a:=simplify(ProbGenerator(MatrixListA));
  b:=simplify(ProbGenerator(MatrixListB));
  eye:=IdentityMatrix(phases);
  oneBar:=Vector(phases,fill=1);
  oneHat:=Vector[rows](phases,fill=1);
  print("a(z)=",a(z));
  print("b(z)=",b(z));
  TGE:=z->phi(z)*(eye*z-a(z))+pi[0]*(eye*z-b(z));
end proc:

#####
#####
#####          Random Walk Generating Functions          #####
#####
#####

### WheelerWhirlwind
#
# Purpose: Generates a random walk rule of (levelSpan) matrices with
#           dimension (phases x phases) using the "Wheeler Whirlwind" algorithm.
#
# NOTE: This is the most random rule generation algorithm in this file, and
#        is the only algorithm that should be used to study general trends.
#
WheelerWhirlwind:=proc(levelSpan,phases)
  local fillingLeft,choiceLS,choiceP,row,toRemove,entree,swig,notPickedLS,
  notPickedP,ratton,rattonLeft,tempy,ret;

  ret:=[seq(Matrix(phases),x=1..levelSpan)];

  choiceLS:=rand(1..levelSpan)();
  choiceP:=rand(1..phases)();

```

```

for row to phases do
  fillingLeft:=1;
  notPickedLS:=[seq(x,x=1..levelSpan)];
  for tempy to levelSpan do

    toRemove:=rand(1..nops(notPickedLS))();
    choiceLS:=notPickedLS[toRemove];
    notPickedLS:=[op(notPickedLS[1..toRemove-1]),
      op(notPickedLS[toRemove+1..nops(notPickedLS)])];

    if tempy<>levelSpan then
      ration:=(rand(0..fillingLeft*(10^7)))/(10^7);
    else
      ration:=fillingLeft;
    end if;
    fillingLeft:=fillingLeft-ration;

    rationLeft:=ration;
    notPickedP:=[seq(x,x=1..phases)];
    for entree to phases do
      toRemove:=rand(1..nops(notPickedP))();
      choiceP:=notPickedP[toRemove];
      notPickedP:=[op(notPickedP[1..toRemove-1]),
        op(notPickedP[toRemove+1..nops(notPickedP)])];

      if entree<>phases then
        swig:=rand(0..rationLeft*(10^7)))/(10^7);
        rationLeft:=rationLeft-swig;
      else
        swig:=rationLeft;
      end if;
      ret[choiceLS][row,choiceP]:=swig;
    end do;
  end do;
end do;
ret:=evalf(ret);
end proc:

```

```

### WeightedMatrixList
#
# Purpose: Generates a random walk rule of (levelSpan) matrices with
#         dimension (phazors x phazors) using the "Weighted Matrix List"
#         algorithm.
#
# NOTE: The rule generated by this function contains matrices whose
#       rows are equal to one another. The expected step of the
#       system is roughly proportional to "drift", which should
#       always be a real number between -100 and 100
#
WeightedMatrixList:=proc(levelSpan,phazors,drift)
  local Template,Nudge,filler,nudger,losers,littleNudge,rightSideFiller,
        natSum,Magic,nthPartialSumHarmonicSeries;

  global DriftA;
  Template:=[0,0];
  #natSum:=((levelSpan-2)*(1+(levelSpan-2)))/2;
  nthPartialSumHarmonicSeries:=0;
  for losers to (levelSpan-2) do
    nthPartialSumHarmonicSeries:=nthPartialSumHarmonicSeries+1/losers;
  end do;

  Magic:=Matrix([[ -1,levelSpan-2,0],[2*phazors,
        phazors*nthPartialSumHarmonicSeries,1-abs(drift/100)]]);

  print(Magic);
  Magic:=ReducedRowEchelonForm(Magic);
  print(Magic);
  filler:=Magic[1,3];
  rightSideFiller:=Magic[2,3];
  nudger:=abs(drift/100);
  Template[1]:=Matrix(phazors,phazors,fill=filler);
  Template[2]:=Matrix(phazors,phazors,fill=filler);
  for losers from 3 to levelSpan do
    Template:=[op(Template),Matrix(phazors,phazors,
        fill=rightSideFiller/(losers-2))];
  end do;
  if (drift>0) then
    for losers to (levelSpan-2) do
      littleNudge:=(StochasticMatrix(phazors,phazors)*nudger)/(levelSpan-2);
      Template[losers+2]:=Template[losers+2]+littleNudge;
    end do;
  end if;
  if (drift<0) then
    Nudge:=StochasticMatrix(phazors,phazors)*nudger;
    Template[1]:=Template[1]+Nudge;
  end if;
  #print(filler);
  #print(rightSideFiller);
  DriftA:=evalf(copy(Template));
end proc:

```

```

#####
#####
#####              Basic Setup Functions              #####
#####
#####              The functions in this section call the rule
#####              generation functions and assign their output
#####              to global variables for convenience.
#####
#####
#####

### WhirlwindRandomWalk
#
# Purpose: Generates a random walk rule using the "Wheeler Whirlwind" algorithm,
#           and a randomly generated boundary rule.
#
WhirlwindRandomWalk:=proc(levelSpan, phazes)
  global MatrixListA,MatrixListB, phases;
  phases:=phazes;
  MatrixListA:=WheelerWhirlwind(levelSpan,phazes);
  MatrixListB:=ProbList(levelSpan-1,phazes);
end proc:

### RandomRandomWalk
#
# Purpose: Generates a randomly generated random walk rule,
#           and a randomly generated boundary rule.
#
RandomRandomWalk:=proc(levelSpan, phazes)
  global MatrixListA,MatrixListB, phases;
  phases:=phazes;
  MatrixListA:=ProbList(levelSpan,phazes);
  MatrixListB:=ProbList(levelSpan-1,phazes);
end proc:

### WeightedRandomWalk
#
# Purpose: Generates a random walk rule using the WeightedMatrixList function,
#           and a randomly generated boundary rule.
#
WeightedRandomWalk:=proc(levelSpany, phazes,drift)
  global MatrixListA,MatrixListB, phases;
  phases:=phazes;
  MatrixListA:=WeightedMatrixList(levelSpany,phases,drift);
  MatrixListB:=ProbList(levelSpany-1,phases);
  #print("BOO-YAH!");
end proc:

```

```

#####
#####
#####          Variable Calculation Functions          #####
#####
#####
#####

### CalcG
#
# Purpose: Calculates a stochastic G matrix that
#          satisfies  $G=a(G)$ .
#
CalcG:=proc(matrixList, iterations) local func, ans, x;
  func:=GFuncGenerator(matrixList);
  ans:=Matrix(phases);
  for x from 1 to iterations
  do
    ans:=simplify(func(ans));
    MakeStochastic(ans);
  end do;
  MakeStochastic(ans);
end proc:

### CalcGSubStochastic
#
# Purpose: Calculates a sub-stochastic G matrix which
#          satisfies  $G=a(G)$ .
#
CalcGSubStochastic:=proc(matrixList, iterations) local func, ans, x;
  func:=GFuncGenerator(matrixList);
  ans:=Matrix(phases);
  for x from 1 to iterations
  do
    ans:=simplify(func(ans));
  end do;
  ans;
end proc:

### CalcPhaseDistribution
#
# Purpose: Calculates the stabilizing phase distribution
#          Pi that satisfies  $Pi.a(1)=Pi$ 
#
CalcPhaseDistribution:=proc(matList) local ret, a1, workmat;
  a1:=simplify(a(1));
  workmat:=Transpose(eye-a1);
  print(workmat);
  ret:=Column(BollingerWheelerRREF(workmat),phases);
  print(BollingerWheelerRREF(workmat,5));
  ret:=-ret;
  ret[phases]:=1;
  ret:=ret/ColumnVectorSum(ret);
  ret;
end proc:

```

```

### CalcDrift
#
# Purpose: Calculates the expected step of the random walk.
#
CalcDrift:=proc(matList) local ret, aDiff, oneBar;
  oneBar:=Vector(phases,fill=1);
  aDiff:=z->add(matList[x+1]*x*z^(x-1),x=1..nops(matList)-1);
  ret:=simplify(aDiff(1)).oneBar;
  ret:=ret-simplify(a(1)).oneBar;
  ret:=Transpose(CalcPhaseDistribution(matList)).ret;
  ret;
end proc:

FindPiZero:=proc(listylistA,listylistB) local G, GFuncB,matty,piZero,i;
  G:=CalcG(listylistA,200);
  GFuncB:=GFuncGenerator(listylistB);
  matty:=BollingerWheelerRREF(Transpose(eye-GFuncB(G)));
  piZero:=Vector(phases);
  for i to phases-1 do
    piZero[i]:=-matty[i,phases];
  end do;
  piZero[phases]:=1;
  piZero:=piZero/ColumnVectorSum(piZero);
  piZero;
end proc:

### FindL
#
# Purpose: Calculates the right eigenvector of GMin.
#
FindL:=proc(lilG) local rho,L;
  rho:=BiggestEVal(lilG);
  L:=Column(BollingerWheelerRREF(Transpose(lilG-rho*eye)),phases);
  L:=-L;
  L[phases]:=1;
  L:=L/ColumnVectorSum(L);
  Transpose(L);
end proc:

```

```
#####
#####
#####          Bollinger-Wheeler Reduced Row Echelon Form          #####
#####          #####
#####          #####
#####

### DSubForward
#
# Purpose: Performs forward substitution on a matrix.
#
# NOTE: The "sigDig" variable determines the rounding
#       threshold -- any entries smaller than  $1 \cdot 10^{-\text{sigDig}}$ 
#       will be rounded to zero.
#
DSubForward:=proc(matty::Matrix,sigDig) local y,x,i,xSize,ySize,bigRow,mat;
  mat:=copy(matty);

  xSize:=ColumnDimension(mat);
  ySize:=RowDimension(mat);
  for x to xSize do;
    #Find the largest leading entry in this column:
    bigRow:=x;

    # If there are more columns than rows, we may be done
    if(x>ySize) then break; end if;
    for y from x+1 to ySize do;
      if (abs(mat[y,x])>abs(mat[bigRow,x])) then bigRow:=y; end if;
    end do;
    if(mat[bigRow,x]=0) then next; end if;

    #Move the 'largest' row so the leading entry is on the diagonal:
    RowOperation(mat,[x,bigRow],inplace=true);

    #Round entries absurdly close to zero:
    for i from x to xSize do;
      if (abs(mat[x,i])< $1 \cdot 10^{-\text{sigDig}}$ ) then mat[x,i]:=0; end if;
    end do;

    if(mat[x,x]<>0) then
      #Scale this row so the leading entry is 1:
      RowOperation(mat,x,1/mat[x,x],inplace=true);
    end if;

    #Kill the leading entries in the rows below this one:
    for y from x+1 to ySize do;
      RowOperation(mat,[y,x],-mat[y,x], inplace=true);

      #Make sure we /really/ zero it out:
      if (abs(mat[y,x])< $1 \cdot 10^{-\text{sigDig}}$ ) then mat[y,x]:=0; end if;
    end do;
  end do;
  mat;
end proc;
```

```

### DSubBackward
#
# Purpose: Performs backwards substitution on a matrix.
#
# NOTE: The "sigDig" variable determines the rounding
#        threshold -- any entries smaller than  $1 \cdot 10^{-\text{sigDig}}$ 
#        will be rounded to zero.
#
DSubBackward:=proc(matty::Matrix,sigDig) local y,x,i,xSize,ySize,mat;
  mat:=copy(matty);
  xSize:=ColumnDimension(mat);
  ySize:=RowDimension(mat);
  for x from xSize to 1 by -1 do;
    for y from ySize to 1 by -1 do;
      if(mat[y,x]<>0) then
        for i from y-1 to 1 by -1 do;
          RowOperation(mat,[i,y],-mat[i,x]/mat[y,x],inplace=true);
          if(abs(mat[i,x])<1*10-(sigDig)) then mat[i,x]:=0; end if;
        end do;
      end if;
    end do;
  end do;
  mat;
end proc:

```

```

### BollingerWheelerRREF
#
# Purpose: Transforms a matrix into the reduced row echelon form
#          by the Bollinger-Wheeler Reduced Row Echelon Form
#          algorith.
#
# NOTE: This function will accept a second argument, "sigDig,"
#        which defaults to two digits less than Maple's
#        "Digits" global variable.
#
#        The "sigDig" variable determines the rounding
#        threshold -- any entries smaller than  $1 \cdot 10^{-\text{sigDig}}$ 
#        will be rounded to zero.
#
BollingerWheelerRREF:=proc(mat) local ret, sigDig;
  if(nargs=1) then sigDig:=Digits-2; end if;
  if(nargs=2) then sigDig:=args[2]; end if;
  ret:=DSubForward(mat,sigDig);
  ret:=DSubBackward(ret,sigDig);
  RoundAwesome(ret,sigDig);
end proc:

```

```

##### Create the Repository #####
end module;
march('create',cat(currentdir(),"wheelbox.mla"));
savelibname:=cat(currentdir(),"wheelbox.mla");
savelib('RandomWalk');

```

```

#####
##                                                                 ##
##                                                                 ##
##                                                                 ##
##              SIMULATIONS                                       ##
##                                                                 ##
##          Copyright 2006 David Wheeler and Drew Bollinger      ##
##                                                                 ##
##                                                                 ##
##                                                                 ##
#####

#####
##                                                                 ##
##              Verification of GMin                               ##
##      This program produces dots which begin at level 1, phase i and ##
##      move accoring to a(z). We examine the probability that a dot ##
##      hits zero at phase j and compare our simulated probability to ##
##      the value predicted by the GMin matrix.                   ##
##                                                                 ##
#####
restart;
kernelopts(gcfreq=10^7);
libname:=libname,catt(currentdir(),"/wheelbox.mla");
with(RandomWalk);
with(LinearAlgebra):
randomize();
Digits:=30;
WhirlwindRandomWalk(5,3);
Housekeeping();
xSize:=100; ySize:=phases;
Cabinet:=Matrix(phases);
RND:=rand(1..9999999)/10000000.;
RunSim:=proc(maxRuns, maxSteps) global bigValue, totalVector, Cabinet, startVector;
  local outputFreq, MatrixListCount, runs, dotX, dotY, startingPhase, step, thresh,
    counter, complete, indexy, newPhase;
  Cabinet:=Matrix(phases);
  totalVector:=Vector(phases);
  outputFreq:=floor(maxRuns/100);
  startVector:=Vector(phases);
  MatrixListCount:=nops(MatrixListA);
  for runs to maxRuns do
    dotX:=2;
    dotY:=rand(1..phases)();
    startingPhase:=dotY;
    startVector[startingPhase]:=startVector[startingPhase]+1;

```

```

for step to maxSteps do
  thresh:=RND();
  counter:=0;
  complete:=0;
  if (dotX>(maxSteps/2)+2) then break; end if;
  if (dotX>1) then
    for indexy to MatrixListCount do
      for newPhase to phases do
        counter:=counter+MatrixListA[indexy][dotY,newPhase];
        if (counter>=thresh) then
          complete:=1;
          dotX:=dotX+(indexy-2);
          dotY:=newPhase;
          break;
        end if;
      end do;
      if (complete=1) then break; end if;
    end do;
  else
    Cabinet[startingPhase,dotY]:=Cabinet[startingPhase,dotY]+1;

    ### number of times the dot hits zero starting from each phase
    totalVector[startingPhase]:=totalVector[startingPhase]+1;
    break;
  end if;
  #print(dotX,dotY);
end do;
if(runs mod outputFreq=0) then print(evalf[3](runs/maxRuns)); end if;
end do;
end proc:
Runs:=100; Steps:=500;
stepVector:=Vector([100,250,500,1000]);
errorMatrix:=Matrix(4);
GMin:=CalcGSubStochastic(MatrixListA,500);
for runs to 4 do
  Runs:=10^(runs+1);
  for steps to 4 do
    Steps:=stepVector[steps];
    RunSim(Runs,Steps);
    for phazors to phases do
      MaybeG[phazors,1..phases]:=copy(Cabinet[phazors,1..phases])/startVector[phazors];
    end do;
    errorMatrix[runs,steps]:=oneHat.(abs(GMin-MaybeG)).oneBar;
  end do;
end do;
errorMatrix:
Cabinet;
GMin:=CalcGSubStochastic(MatrixListA,300);

```

```
#####
##
##           Verification of Expected Step           ##
##   This program begins a dot at any point in S and allows it to   ##
##   move according to a(z). After the simulation, we calculate       ##
##   the average "velocity" of the dot and compare it the expected   ##
##   step.                                                           ##
##                                                                 ##
#####

restart;
libname:=libname,catt(currentdir(),"/wheelbox.mla");
with(RandomWalk);
with(plots);
with(plottools);
with(LinearAlgebra):
randomize();
RND:=rand(1..99999999)/100000000.;
WhirlwindRandomWalk();
Housekeeping();
xSize:=100; ySize:=phases;
Sandbox:=Matrix(ySize,xSize);
phaseDistro:=Vector(phases);

RunSim:=proc(maxRuns, maxSteps)
  local runs,dotX,dotY,step,thresh,counter,complete,indexy,newPhase,outputFreq;
  global Sandbox, bigValue, total, pointList, phaseDistro;
  Sandbox:=Matrix(ySize,xSize);
  pointList:=Matrix(maxRuns,maxSteps);
  total:=0;
  outputFreq:=floor(maxRuns/100);
  for runs to maxRuns do
    #dotX:=rand(2..xSize)();
    #dotY:=rand(1..ySize)();
    dotX:=2000;
    dotY:=rand(1..phases)();
    bigValue:=0;
    for step to maxSteps do #while (dotX<=xSize and dotY<=ySize) do
      total:=total+1;
      pointList[runs,step]:=[dotX,dotY];
      phaseDistro[dotY]:=phaseDistro[dotY]+1;

      if (dotX<=xSize and dotY<=ySize) then
        Sandbox[dotY,dotX]:=Sandbox[dotY,dotX]+1;
        if(Sandbox[dotY,dotX]>bigValue) then bigValue:=Sandbox[dotY,dotX];
        end if;
      end if;

      thresh:=RND();
      counter:=0;
      complete:=0;
    end do;
  end do;
end proc;
```

```

if (dotX>1) then
  for indexy from 1 to nops(MatrixListA) do
    for newPhase from 1 to phases do
      counter:=counter+MatrixListA[indexy][dotY,newPhase];
      if (counter>=thresh) then
        complete:=1;
        dotX:=dotX+(indexy-2);
        dotY:=newPhase;
        break;
      end if;
    end do;
    if (complete=1) then break; end if;
  end do;
else
  for indexy from 1 to nops(MatrixListB) do
    for newPhase from 1 to phases do
      counter:=counter+MatrixListB[indexy][dotY,newPhase];
      if (counter>=thresh) then
        complete:=1;
        dotX:=dotX+(indexy-1);
        dotY:=newPhase;
        break;
      end if;
    end do;
    if (complete=1) then break; end if;
  end do;
end if;

#print(dotX,dotY);
end do;
if(runs mod outputFreq=0) then print(evalf[3](runs/maxRuns)); end if;
end do;
end proc:

VelocityAvg:=proc(mat) local plotList,run,stepList,numSteps,numRuns,step, old,
new, xScale,yScale, accStep, avgVel, runVel;
#accStep stands for 'accuracy step' -- it determines how many line segments
#get plotted
if(nargs=2) then accStep:=args[2]; else accStep:=2; end if;
plotList=[];
xScale:=.5;
yScale:=2;

runVel:=0;
avgVel:=0;

numRuns:=RowDimension(mat);
for run to numRuns do
  stepList:=Row(mat,run);
  old:=[1,(stepList[2][1]-stepList[1][1])*yScale];
  numSteps:=Dimension(stepList);

```

```

for step from 2*accStep to numSteps by accStep do
  #if(step mod accStep=0) then
    new:=[step*xScale,((stepList[step][1]-stepList[step-accStep][1])/accStep)
    *yScale];
    #plotList:=[op(plotList),line(old, new, color=black)];
    old:=new;
  #end if;
end do;
runVel:=(stepList[numSteps][1]-stepList[2][1])/numSteps;
printf("Avg Velocity for run %d: %f\n",run,runVel);
avgVel:=avgVel+runVel;
end do;
avgVel:=avgVel/numRuns;
printf("Overall Avg Velocity: %f\n",avgVel);
end proc:

Runs:=10000; Steps:=1000;
RunSim(Runs,Steps);
VelocityAvg(pointList,40);#round(Runs*Steps/2000)+1);
CalcDrift(MatrixListA);

#####
##
##                               Verification of Proposition 1
##                               ##
##   This program begins a dot at varying levels (2-5), phase i
##   and allows it to move according to a(z). We record the
##   number of times it hits zero at phase j. Finally, we normailze
##   the simulated matrix and compare it to oneBar*L.
##                               ##
##                               ##
#####
restart;
kernelopts(gcfreq=10^7);
libname:=libname,catt(currentdir(),"/wheelbox.mla");
with(RandomWalk);
with(LinearAlgebra):
randomize();
Digits:=30;
WhirlwindRandomWalk(3,3);
temp:=CalcDrift(MatrixListA);
while temp>.01 or temp<0 do
  MatrixListA:=WheelerWhirlwind(3,3);
  temp:=CalcDrift(MatrixListA);
end do;
Housekeeping();
xSize:=100; ySize:=phases;
Cabinet:=Matrix(phases,phases);
RND:=rand(1..9999999)/10000000.;
RunSim:=proc(maxRuns, maxSteps, startLevel) global bigValue, total, Cabinet;
  local outputFreq, MatrixListCount, runs, dotX, dotY, startingPhase, step,
  thresh, counter, complete, indexy, newPhase;
  Cabinet:=Matrix(phases);
  total:=0;
  outputFreq:=floor(maxRuns/100);

```

```

MatrixListCount:=nops(MatrixListA);
for runs to maxRuns do
  dotX:=startLevel;
  dotY:=rand(1..phases());
  startingPhase:=dotY;
  for step to maxSteps do

    thresh:=RND();
    counter:=0;
    complete:=0;
    if (dotX>(maxSteps/2)+2) then break; end if;
  if (dotX>1) then
    for indexy to MatrixListCount do
      for newPhase to phases do
        counter:=counter+MatrixListA[indexy][dotY,newPhase];
        if (counter>=thresh) then
          complete:=1;
          dotX:=dotX+(indexy-2);
          dotY:=newPhase;
          break;
        end if;
      end do;
      if (complete=1) then break; end if;
    end do;
  else
    Cabinet[startingPhase,dotY]:=Cabinet[startingPhase,dotY]+1;
    total:=total+1; ### number of times the dot hits zero
    break;
  end if;
  #print(dotX,dotY);
end do;
if(runs mod outputFreq=0) then print(evalf[3](runs/maxRuns)); end if;
end do;
end proc:
Runs:=10000; Steps:=500;
GMin:=CalcGSubStochastic(MatrixListA,400);
L:=FindL(GMin);
runVector:=3*Vector([1500,5900,22400,85700]);
errorVector:=Vector(4);
for level from 5 to 6 do
  Runs:=runVector[level-2];
  RunSim(Runs,Steps,level);
  print(MakeStochastic(copy(Cabinet)));
  errorVector[level-2]:=oneHat.abs((oneBar.L)-
    (MakeStochastic(copy(Cabinet))))).oneBar;
end do;
Cabinet;
MakeStochastic(copy(Cabinet));
errorVector;

```

```
#####
##
##                               Verification of Proposition 2                               ##
##      This program begins a dot at level 1, phase i with probability                    ##
##      l_i and allows it to move according to a(z). We record how many                ##
##      times the dot hits zero compared to the total number of runs.                  ##
##      This program shows that as the number of runs and steps approach              ##
##      infinity, the ratio of "zero hits" to runs approaches rho.                    ##
##                                                                                       ##
#####
restart;
kernelopts(gcfreq=10^7);
libname:=libname,catt(currentdir(),"/wheelbox.mla");
with(RandomWalk);
with(LinearAlgebra):
randomize();
Digits:=30;
WhirlwindRandomWalk(5,3);
Housekeeping();
xSize:=100; ySize:=phases;
Cabinet:=Matrix(phases,phases);
RND:=rand(1..9999999)/10000000.;
RunSim:=proc(maxRuns, maxSteps) global bigValue, total, Cabinet;
  local outputFreq, MatrixListCount, runs, dotX, dotY, otherThresh,
  otherCounter, otherComplete, yChoice, startingPhase, step, thresh,
  counter, complete, indexy, newPhase;
  total:=0;
  outputFreq:=floor(maxRuns/100);
  MatrixListCount:=nops(MatrixListA);
  for runs to maxRuns do
    dotX:=2;
    ##dotY selection cycle
    otherThresh:=RND();
    otherCounter:=0;
    otherComplete:=0;
    for yChoice to phases do
      otherCounter:=otherCounter+trialsMatrix[yChoice];
      if (otherCounter>=otherThresh) then
        otherComplete:=1;
        dotY:=yChoice;
        break;
      end if;
    end do;
    startingPhase:=dotY;
    for step to maxSteps do

      thresh:=RND();
      counter:=0;
      complete:=0;
```

```

if (dotX>(maxSteps/2)+2) then break; end if;
if (dotX>1) then
  for indexy to MatrixListCount do
    for newPhase to phases do
      counter:=counter+MatrixListA[indexy][dotY,newPhase];
      if (counter>=thresh) then
        complete:=1;
        dotX:=dotX+(indexy-2);
        dotY:=newPhase;
        break;
      end if;
    end do;
    if (complete=1) then break; end if;
  end do;
else
  Cabinet[startingPhase,dotY]:=Cabinet[startingPhase,dotY]+1;
  total:=total+1; ### number of times the dot hits zero
  break;
end if;
#print(dotX,dotY);
end do;
if(runs mod outputFreq=0) then print(evalf[3](runs/maxRuns)); end if;
end do;
end proc;
Runs:=1000; Steps:=1000;
Gmin:=CalcGSubStochastic(MatrixListA,400);
rho:=BiggestEval(Gmin);
trialsMatrix:=FindL(Gmin);
stepVector:=Vector([100,250,500,1000]);
errorMatrix:=Matrix(4); ##abs of rho-(total/runs)
for runs to 4 do
  Runs:=10^(runs+1);
  for steps to 4 do
    Steps:=stepVector[steps];
    RunSim(Runs,Steps);
    errorMatrix[runs,steps]:=abs(rho-(total/Runs));
  end do;
end do;
total;
rho:=BiggestEval(Gmin);
errorMatrix;

```

REFERENCES

- [1] D. E. Barrett, H. R. Gail, S. L. Hantler, B. A. Taylor, *Varieties in a Two Dimensional Polydisk with Univalent Projection at the Boundary*, IBM research report RC15846, 1990.
- [2] Guy Fayolle, Roudolf Iasnogorodski, Vadim Malyshev, *Random Walks in the Quarter-Plane*, Springer, Berlin, 1999.
- [3] Gregory F. Lawler, Lester N. Coyle, *Lectures on Contemporary Probability* American Mathematical Society, Providence, 1999.

E-mail address, Drew Bollinger: drewbo@umich.edu

E-mail address, David Wheeler: dwwheel@umich.edu

214 PACKARD ST., ANN ARBOR, MI 48109