AN EXPERIMENTAL APPROACH TO STUDYING RAMANUJAN GRAPHS

Kevin Chang

Math Junior Seminar

May 26, 2001

INTRODUCTION

The explicit construction of certain sparse and highly connected graphs, called expander graphs, has many important applications, especially in networking. Of special interest are the Ramanujan graphs, which are a subset of the expanders. Explicit constructions of families of Ramanujan graphs have been discovered, but are very complicated. It is conjectured, however, that the very simple set of random 3-regular bipartite graphs may be just as good as the explicit constructions in the limit as the number of vertices in the graphs approaches infinity. This paper shows the results of experiments to determine whether this conjecture is consistent with numerical results. It is presented in six sections. The theory section defines Ramanujan graphs, states some of their useful properties, discusses construction of such graphs, and introduces the hypotheses of my experiments. The methods section describes the two main algorithms used in the experiments and the approximate amount of computing resources they need. The procedures section describes the actual structure of my experiments. The results section gives the data derived from the experiments and the discussion gives my analysis of the results.

THEORY

Preliminaries

For this paper, we shall consider undirected graphs $X = (V, E)$ where V is the set of vertices and E is the set of edges of X. We define the **expander constant** of X as the largest constant $h(X)$ s.t. $|\partial A| \geq h(X)|A|$ for all subsets A of V, where ∂A is the boundary of A, that is the set of vertices v in V-A such that there is an edge between v and some vertex in A.

A graph with a vertex set of size n can be represented as an n x n matrix called the **adjacency matrix**. Each row and column corresponds to a vertex in the graph. For the adjacency matrix $B = \{b_{ij}\}$, the element b_{ij} is the multiplicity of the edges between vertices i and j. Note that B is always a real, symmetric matrix. We call the n eigenvalues of the matrix, $\lambda_0, \lambda_{1,\ldots,} \lambda_{n-1}$ in descending order, the spectrum of the graph.

In my experiments, I work exclusively with **k–regular bipartite graphs**.
A graph X is **bipartite** if V can be partitioned into two disjoint sets such that no edge is incident to two vertices in the same set. X is **k-regular** if each vertex has exactly k edges incident to it. For bipartite k-regular graphs, we know additionally that $\lambda_0 = k = -\lambda_{n-1}$ and that $\lambda_i = -\lambda_{n-i-1}$, for all i. From this point on, X will be a k-regular bipartite graph.

FACT 1: If $h(X)$ is the expander constant of a graph X, then

$$\frac{k-\lambda_1}{2} \leq h(X) \leq \sqrt{2k(k-\lambda_1)}.$$ [DSV]

We call the value k- λ_1 the **spectral gap**. A larger spectral gap insures us a higher lower bound on the expander constant. An expander with a larger $h(X)$ is considered to be of "higher quality."

FACT 2: Asymptotically, the size of the spectral gap is bounded, for the size of the second largest eigenvalue is lower bounded by the inequality $\lim_{n\to\infty} \lambda_{1n} \geq 2\sqrt{2k-1}$, where $\{\lambda_{1n}\}$ is a sequence of eigenvalues derived from any sequence of graphs whose size is going to infinity. [Sar]

This observation motivates the following definition of a Ramanujan graph, which in the limit is a graph with the largest spectral gap possible.
We define a graph to be **Ramanujan** if $\lambda_1 \leq 2\sqrt{k-1}$.

Why we care about Ramanujan graphs

In this section, we allow X to be any k-regular graph, relaxing the bipartite condition. Such graphs have the useful properties of small diameters, large chromatic numbers, and small independence numbers. Let μ_1 be the second-largest eigenvalue in absolute value; note that if the graph is bipartite, we have $\mu_1 = \lambda_1$.

An **independent set** of X is a set $I \subseteq V$ such that none of the vertices in I share an edge. The **independence number** of X, i(X), is the size of the largest independent set in X.

FACT 3: The independence number is bounded from above by $i(X) \leq \frac{n}{k}\mu_1$. [Sar]

A **coloring** of X is a labeling of all vertices of C in such a way that no two vertices that share an edge are labeled with the same color. The **chromatic number** of X is the coloring of X that uses the least number of different colors. Note that if X is bipartite, its chromatic number is two.

FACT 4: The chromatic number is bounded from below by $\chi(X) \geq \frac{k}{\mu_1}$. [Sar]

The **diameter** of X is the longest path between any two vertices in V.

FACT 5: The diameter is bounded from above by

$$diam(X) \leq \frac{\log(2n)}{\log\left(\frac{k+\sqrt{k^2-\mu_1^2}}{\mu_1}\right)}. \quad \text{[Sar]}$$

From Facts 3, 4, and 5, we conclude that Ramanujan graphs are in a sense optimal (based at least on the bounds we know about presently; these bounds may not be tight). Since they have minimally sized λ_1 in the limit, they have a minimally sized lower bound for μ_1 (since $\mu_1 \geq \lambda_1$); they thus have a minimally sized upper bound on the independence number and diameter and a maximally sized lower bound on the chromatic number. The

fact that Ramanujan graphs are sparse but have small diameters makes them very useful for such real world applications as network building.

On Constructing Ramanujan graphs

A **family** of Ramanujan graphs is an infinite sequence of graphs, whose size is going to infinity, and whose graphs satisfy the Ramanujan criteria.

FACT 6: For $k = 3$, $k = p+1$, and $k = q+1$ where p is an odd prime and q is a prime power, there are explicit constructions of families of k-regular graphs which are Ramanujan. Refer to [Chi], [LPS], and [Mor] respectively.

A possible alternative to these constructions is the simple example of the random 3-regular bipartite graph.

CONJECTURE 1: With probability 1, if the X_n's are a family of 3-regular bipartite graphs, then $\lambda_1(X_n) \to 2\sqrt{2}$ as $n \to \infty$.

CONJECTURE 1′: With probability 1, 3-regular bipartite graphs are Ramanujan as $n \to \infty$.

If these conjectures are true, then the random graph is just as good as the arithmetically constructed Ramanujan graph.

Another possible alternative to the explicit constructions is the random 7-regular bipartite graph. $k = 7$ is an especially interesting case, because 7 is the first value of k for which there is not an explicit construction (consult FACT 6).

CONJECTURE 2: With probability 1, if the X_n's are 7-regular bipartite graphs, $\lambda_1(X_n) \to 2\sqrt{6}$ as $n \to \infty$.

CONJECTURE 2′: With probability 1, 7-regular bipartite graphs are Ramanujan as $n \to \infty$.

METHODS

In order to test these conjectures, we need a computer algorithm to generate a random k-regular bipartite graph and an algorithm to compute the second largest eigenvalue of the corresponding adjacency matrix. In the appendix, I provide my C++ implementation of these algorithms.

We create a random k-regular bipartite graph by picking edges in k rounds. We first label all n nodes with integers between 0 and n-1. In each round, we randomly permute the nodes labeled $n/2$, $n/2+1 \ldots n-1$ into $r_{n/2}$, $r_{n/2+1}$, …, r_{n-1} and insert all edges of the form

(i, $r_{i+n/2}$) into the graph. After one round, each node has exactly one edge incident to it and edges go only between the set of nodes with labels smaller than $n/2$ and the set of nodes with labels larger than $n/2$. Thus, after k rounds of inserting edges, we have randomly chosen a k-regular bipartite graph. Clearly this method chooses graphs from the set of k-regular bipartite graphs (where the biparition of V is into two sets of size $n/2$; this is necessary for regularity) with a uniform probability density.

This method does not rule out the possibility of the same edge being inserted multiple times into the graph, so called double, triple etc, bonds. Indeed, for arbitrary n and k, it is possible to lower bound the probability of a double bound occurring after two rounds by e^{-1}(using the principle of inclusion-exclusion). For our numerical experiments, it was of interest to eliminate such instances of multiple bonds. In order to do so, we can modify our method of picking a random permutation. In each round, instead of choosing a random permutation of the $n/2$, $n/2+1$,…, $n-1$ nodes in one step, we choose the permutation one node at a time in the following way. If we are presently choosing the random node that will be incident to the node i, we choose randomly from the nodes that have not already been connected to node i. Note that this fixing up will work most of the time, but that it may fail on the last few nodes left to be picked in which case we would start the round over. An alternative of this algorithm is to generate (and discard) random graphs until one is chosen with no multiple bonds. We essentially choose graphs at random until we get "lucky" with a single-bonded graph. For $k = 3$, this discarding method worked beautifully; for $k = 7$, this method proved a disaster, not getting lucky once after many hours of computing. For $k = 7$, the first method did work.

The discarding method ran instantaneously (relative to the eigenvalue computation) for $k = 3$ but did not work at all for $k = 7$. The modified permutation method for $k = 7$ took as long as a second to produce a graph with 700 nodes.

I tried several ways to compute the pertinent eigenvalue of the adjacency matrix. The first method I used involved transforming the n x n adjacency matrix into a tridiagonal matrix, then computing the eigenvalues using the QL algorithm with implicit shifts. [PTVS] However, the computing resources needed for this method proved unsatisfactory since for our purposes we only need the second largest eigenvalue. Furthermore, our graphs are sparse, having only nk edges, and so any algorithm that computes on an n x n representation of the graph will take an unnecessary amount of time and space. Instead, I used the power method using an adjacency list representation of the graph. The power method is an iterative procedure whose successive guesses converge on the eigenvector corresponding to λ_1. Our first guess is any random vector x. Remember that the eigenvectors of a matrix form an orthonormal basis, call them $e_{0,}$…, e_{n-1}. Given our current guess $x = c_0e_0+\ldots+c_{n-1}e_{n-1}$, we first project out the dimensions corresponding to eigenvalues of k and $-k$ (i.e. e_0 and e_{n-1}) and normalize the result, call it y. We then compute $A^2y = \lambda_1{}^2c_1e_1+ \lambda_2{}^2 c_2e_2+\ldots$. A^2y is our new guess and we iterate. Since λ_1 and λ_{n-2} are the largest eigenvalue remaining, successive application of A^2 followed by normalization will have the effect of causing the e_1 (and in our case e_{n-2} as well) components to dominate the others. With this combination of e_1 and e_{n-2}, it is easy to pull out their eigenvalue.

The power method for computing the eigenvalues of a matrix was very fast, taking at most a second to compute the pertinent eigenvalue of a graph with 1000 nodes.

PROCEDURES

Armed with these two algorithms, I conducted 4 different numerical experiments. Experiments 1 and 2 were designed to test Conjectures 1 and 1′ while Experiments 3 and 4 were analogous experiments to test Conjectures 2 and 2′.

EXPERIMENT 1: In order to ascertain the asymptotic behavior of 3-regular bipartite graphs of size n, I generated random 3-regular bipartite graphs with incrementally larger vertex set cardinality and computed their eigenvalues. The different vertex set cardinalities were n = 100, 200, …, 1000. For each n, I sampled the space of random graphs 5000 times.

EXPERIMENT 2: In order to see the effect of eliminating double and triple bonds from experiment 1, I repeated Experiment 1, but with 5000 3-regular bipartite graphs with only single bonds.

EXPERIMENT 3: In order to ascertain the asymptotic behavior of 7-regular bipartite graphs of size n, I generated random 7-regular bipartite graphs with incrementally larger vertex set cardinality and computed their eigenvalues. The different vertex set cardinalities were n = 100, 200, …, 1000. For each n I sampled the space of random graphs 5000 times.

EXPERIMENT 4: In order to see the effect of eliminating multiple bonds from Experiment 3, I repeated Experiment 3 but with 7-regular bipartite graphs with only single bonds.

RESULTS

Number of nodes	mean λ_1	standard deviation of λ_1	% Ramanujan
100	2.807569	0.042382	76.14
200	2.816016	0.026528	76.36
300	2.818708	0.019609	77.38
400	2.821027	0.017542	75.20
500	2.821555	0.014412	76.62
600	2.822497	0.013317	77.54
700	2.822604	0.012812	78.46
800	2.823074	0.011166	79.68
900	2.823307	0.010558	80.34
1000	2.823495	0.009277	79.86

TABLE 1

Table 1 presents the results of Experiment 1. Table 1 displays, for each value of n, the mean value and standard deviation of 5000 samples of the λ_1's of 3-regular bipartite random graphs, allowing double and triple bonds. For each increase in the number of nodes in the graph, the mean λ_1 gets closer to $2\sqrt{2} = 2.82842$ while the standard deviation of the λ_1's gets smaller and smaller. Notice that this is true for each increment without exception.

Number of nodes	mean λ_1	standard deviation of λ_1	% Ramanujan
100	2.775447	0.031092	95.28
200	2.800681	0.018521	93.06
300	2.808354	0.013579	92.84
400	2.812827	0.011292	91.22
500	2.815148	0.009485	91.40
600	2.816813	0.008522	90.90
700	2.818056	0.007648	91.00
800	2.818975	0.00692	90.58
900	2.819689	0.006544	91.06
1000	2.820263	0.005963	91.12

TABLE 2

Table 2 presents the results of Experiment 2. Table 2 displays, for each value of n, the mean value and standard deviation of 5000 samples of the λ_1's of 3-regular bipartite random graphs with only single bonds. For each increase in the number of nodes in the graph, the mean λ_1 gets closer to $2\sqrt{2}$ while the standard deviation of the λ_1's gets smaller and smaller. Notice that this is true for each increment without exception. The percentage of graphs that are Ramanujan appears to initially decrease as n increases and then stabilizes.

Number of nodes	mean λ_1	standard deviation of λ_1	% Ramanujan
100	4.790609	0.113151	83.74
200	4.833492	0.069335	82.68
300	4.848965	0.053038	83.54
400	4.858293	0.043208	82.90
500	4.864893	0.036075	82.92
600	4.868764	0.032218	83.20
700	4.870869	0.028466	84.02
800	4.873587	0.026872	83.18
900	4.875382	0.024967	82.84
1000	4.877063	0.022453	83.92

TABLE 3

Table 3 is the analogue of Table1 for Experiment 3. Notice again that for each increase in the number of nodes in the graph, the mean λ_1 gets closer to $2\sqrt{6} = 4.898979$ while the standard deviation of the λ_1's gets smaller and smaller. Notice that this is true for each increment without exception.

Numberof nodes	mean λ_1	standard deviation of λ_1	% Ramanujan
100	4.530263	0.100469	99.90
200	4.709318	0.062785	99.70
300	4.766999	0.047979	99.42
400	4.796075	0.04044	98.92
500	4.814628	0.035235	98.77
600	4.82776	0.03066	98.26
700	4.836239	0.027734	98.20

TABLE 4

Table 4 is the analogue of Table2 for Experiment 4. Notice again that for each increase in the number of nodes in the graph, the mean λ_1 gets closer to $2\sqrt{6} = 4.898979$ while the standard deviation of the λ_1's gets smaller and smaller. Notice that this is true for each increment without exception. The percentage of graphs that are Ramanujan appears to decrease with n.

DISCUSSION

The data from Experiments 1 and 2 are consistent with Conjecture 1. They show that as n increases, the distribution of the λ_1's of 3-regular bipartite graphs are getting tighter around their mean, as evidenced by the shrinking standard deviation. Furthermore, as n increases, the mean of the λ_1's approaches the magic number $2\sqrt{2}$. Thus, at least up to $n = 1000$, the λ_1's appear to be converging to $2\sqrt{2}$ as n increases.

There is a striking difference between the means and standard deviations of the λ_1's derived from 3-regular bipartite graphs with only single bonds (Experiment 2) and those with multiple bonds allowed (Experiment 1). The mean value of the λ_1's for graphs with only single bonds approaches $2\sqrt{2}$ much more slowly than the mean values of the λ_1's for graphs with multiple bonds. On the other hand, the standard deviation of the λ_1's for graphs with only single bonds are lower than the standard deviations of the λ_1's for graphs with multiple bonds. Thus, the distribution of λ_1's for graphs with single bonds is tighter around its mean than the corresponding distribution of λ_1's for graphs with multiple bonds.

The percentages of multiple-bonded graphs that are Ramanujan appear to be stable with n, all around 78%. For single-bonded graphs, these percentages decrease and then stabilize around 90%. The fact that these percentages are not near 100, indicates that the proportion of 3-regular bipartite graphs that are Ramanujan does not go to one as the size of the vertex set goes to infinity. Thus, our data do not support Conjecture 1′. However, when drawing this conclusion (and all others as well), we must remember the

caveat that we are only checking $n \leq 1000$ and cannot be sure of the behavior for arbitrarily large n.

The data from Experiments 3 and 4 are similar to those of Experiments 1 and 2. The data from Experiments 3 and 4 are consistent with Conjecture 2. They both show that as n increases, the mean of the λ_1's of 7-regular bipartite graphs approaches the number $2\sqrt{6}$ while the distributions of the λ_1's get tighter about this mean. Thus, the λ_1's appear to be converging to $2\sqrt{6}$ as n increases.

Like in Experiment 1, the percentages of multiple-bonded, 7-regular bipartite graphs that are Ramanujan appear to be stable with n, all around 83%. For single-bonded, 7-regular bipartite graphs, this percentage decreases for all n that we checked. Most likely, if we checked for larger n, we would find that this percentage stabilizes like in Experiment 2. Since these percentages do not appear to be approaching 100, these data indicate that the proportion of 7-regular bipartite graphs that are Ramanujan does not go to one as n goes to infinity. Our data thus do not support Conjecture $2'$.

CONCLUSION

I have presented the results of my experiments to determine whether or not the λ_1's of 3- and 7-regular bipartite graphs converge to $2\sqrt{3}$ and $2\sqrt{6}$ respectively, and whether or not such graphs are all less than these magic numbers (i.e. if they are Ramanujan). The results indicate that for both 3- and 7-regular graphs, the λ_1's do appear to converge to the requisite numbers but, converge from both sides of the limit and are thus not Ramanujan in the limit. Furthermore, these results were robust over the choice of single-bonded and multi-bonded graphs.

Further experiments along these lines could involve increasing n and trying different k's. In order to use larger k's for single-bonded graphs, it may be necessary to devise a more efficient graph-generating algorithm.

ACKNOWLEDGEMENTS

Major domo to Steve Miller and Professor Sarnak for their help and guidance for this work. My discussions with Peter Richter on the subject were entertaining and enlightening. Lastly, thanks to my family for their continuous support of my studies and their maintenance of my sanity.

REFERERENCES

[Chi] Chiu, Patrick. "Cubic Ramanujan graphs". *Combinatorica* 12 (1992), no. 3, 275-285.

[DSV] Davidoff, G., Sarnak, P., Valette, A. *An Elementary Approach to Ramanujan Graphs*. preprint, 2001.

[LPS] Lubotzky, A. Phillips, R, Sarnak, P. "Ramanujan graphs" *Combinatorica* 8 (1988), no. 3 261-277.

[Mor] Morgenstern, Moshe. "Existence and explicit constructions of q+1 regular Ramanujan graphs for every prime power q". *J. Combin. Theory Ser. B* (1994) no. 1, 44-62.

[PTVF] Press, W. Teukolsy, S., Vetterling, W., Flannery, B. *Numerical Recipes in C: The Art of Scientific Computing*. 1992, Cambridge University Press, USA.

[Sar] Sarnak, Peter. *Some Applications of Modular Forms*.1990, Cambridge University Press, Cambridge UK.

APPENDIX

The following C++ code was used for all 4 experiments. The function makegraph( ) implements the graph generating algorithm while the function fast( ) implements the power method algorithm for computing the desired eigenvalue. This code works as advertised with the MS Visual Studio C++ compiler and the GNU C++ compiler for UNIX, except for makegraph2( ) which appears to only work properly with MSV C++.

```
#include <math.h>
#include <iostream>
#include <fstream>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
using namespace std;

extern double gasdev(long *idum);

float** makegraph(int n, int k);
void freematrix(int n, float** a);
float powermethod(float ** A, int n, float a, long *idum, int k);

void main()
{
	int k = 7;
	srand( (unsigned)time( NULL ) );

	char filename[20] = "100data.txt";
	for(n = 100; n < 1001; n+=100)
	{
		ofstream fout;
		int regular = 0, ramanujan = 0;
		// counters to keep track of number of regular and ramanujan
		fout.open(filename);
		fout<<n<<"  nodes"<<endl;
		for(int z = 0; z < 5000; z++)
		{
			float ** A = makegraph(n, k);
			float eigen = powermethod(A, n, .00001, &idum, k);
			fout<<eigen<<'\t';
			if(eigen <=2*sqrt(2))
				ramanujan++;
			regular++;
			fout<<regular<<'\t'<<ramanujan<<endl;
			freematrix(n, A);
		}
		filename[0]++;
		fout.close();
	}
}

float** makematrix(int n, int k)
//return an nxk matrix of floats
{
	float** graph;
	graph = (float**)calloc(n, sizeof(int*));
	for(int i = 0; i < n; i++)
		graph[i] = (float*)calloc(k,sizeof(float));
	return graph;
}

int compare(const void* e1, const void* e2)
//comparison for call to qsort in makegraph
{
	int* v1 = *(int**)(e1), *v2 = *(int**)(e2);
```

```
        if(v1[0] < v2[0]) return -1;
        if(v1[0] == v2[0]) return 0;
        return 1;
}

int comparefloat(const void* e1, const void* e2)
{
        float v1 = *(float*)(e1), v2 = *(float*)(e2);
        if(v1 < v2) return 1;
        if(v1 == v2) return 0;
        return -1;
}

void freematrix(int n, float** a)
//free a matrix defined by makematrix()
{
        for(int i = 0; i < n; i++)
                free(a[i]);
        free(a);
}

float** makegraph(int n, int k)
//makes a random k-regular bipartite graph of size n
{
        int i, j; float** graph;
        if(n%2 != 0)
                fprintf(stderr, "n needs to be EVEN\n");
        graph = makematrix(n, n);
        float** b = makematrix(n/2, 2);
        bool flag = 0;
        while(flag == 0)
        {
                for(j = 0; j < k; j++)
                {
                        for(i = 0; i < n/2; i++)
                        {
                                b[i][0] = rand();
                                b[i][1] = n/2+i;
                                //assign each node label between n/2 and n a random number
                                //stick everything in b
                        }
                        qsort((void*)b, n/2, sizeof(int*), compare);
                        // sort b based on the rand's.node labels
                        // along for the ride" are put into random order
                        for(i = 0; i < n/2; i++)
                                // insert these random edges into the graph
                                // represented by an adjacency list.
                        {
                                graph[i][j] = b[i][1];
                                graph[(int)b[i][1]][j] = i;
                        }
                }
                flag = 1;
        // remove the comments below if you only want single bonds, using the throw away
        method
        /*      for(i = 0; i < n; i++)
                        for(j = 0; j < k-1; j++)
                        {
                                float test = graph[i][j];
                                for(int l = j+1; l < k; l++)
                                        if(graph[i][l] == test)
                                                flag = 0;
                        }
                if(flag == 0)
                        for(i = 0; i < n;i++)
                                for(j = 0; j < k; j++)
                                        graph[i][j] = 0;*/

        }
                freematrix(n/2, b);
                return graph;
```

```
}

float powermethod(float** A, int n, float tolerance, long* idum, int k)
{
  //our current and previous guess for the eigenvalue
  float guess1 = 1, guess2 = 10;
  //our guess for the eigenvector
  float *eigenvec1 = (float*)calloc(n, sizeof(float));
  //vectors to work with
  float *eigenvec2 = (float*)calloc(n, sizeof(float));
  float *vec3 = (float*)calloc(n, sizeof(float));
  //initialize our first guess
  for(int i = 0; i < n; i++)
    eigenvec1[i] = (float)rand()/RAND_MAX;

  while(guess1-guess2 <= -tolerance || guess1-guess2 >= tolerance)
        {

                float c = 0, d = 0, normconst = 0,test = 0;
                // set the value of c and zero out vec3
                for(i = 0; i < n; i++)
                {
                  c += eigenvec1[i];
                  vec3[i] = 0;
                }
                c = c/n;
                //set the value of d
                for(i = 0; i < n/2; i++)
                        d += eigenvec1[i];
                for(i = n/2; i < n; i++)
                        d -= eigenvec1[i];
                d = d/n;
                //set the value of eigenvec2
                for(i = 0; i < n/2; i++)
                        eigenvec2[i] = eigenvec1[i]-c-d;
                for(i = n/2; i < n; i++)
                        eigenvec2[i] = eigenvec1[i]-c+d;
                //normalize eigenvec2 with L2 norm
                for(i = 0; i < n; i++)
                        normconst += eigenvec2[i]*eigenvec2[i];
                normconst = sqrt(normconst);
                for(i = 0; i < n; i++)
                        eigenvec2[i] = eigenvec2[i]/normconst;
                //compute A(A(eigenvec2)), place results in eigenvec1.
                for(i = 0; i < n; i++)
                        for(int j = 0; j < k; j++)
                            vec3[i] += eigenvec2[(int)A[i][j]];
                for(i = 0; i < n; i++)
                        eigenvec1[i] = 0;
                for(i = 0; i < n; i++)
                        for(int j = 0; j < k; j++)
                            eigenvec1[i] += vec3[(int)A[i][j]];
                guess2 = guess1;
                guess1 = 0;
                //compute eigenvalue guess that corresponds to our
                //eigenvector guess
                for(i = 0; i < n; i++)
                        guess1 += eigenvec2[i]*eigenvec1[i];
                guess1 = sqrt(guess1);
   }
  free(eigenvec1);
  free(eigenvec2);
  free(vec3);
  return guess1;
}

float** makegraph2(int n, int k)
//makes a random single-bonded k-regular bipartite graph of size n using the smarter
//permutation.  Appears to only work correctly with MSV C++.
{
        int i, j; float** graph;
```

```
        if(n%2 != 0)
                fprintf(stderr, "n needs to be EVEN\n");
        graph = makematrix(n, n);
        float** b = makematrix(n/2, 2);
        int* nodes = (int*)calloc(n/2, sizeof(int));
        for(j = 0; j < k; j++)
        {
                int bad;
                for(i = 0; i < n/2; i++)
                        nodes[i] = n/2+i;
                for(i = 0; i < n/2; i++)
                {
                        bad = 1;
                        while(bad == 1)
                        {
                                bad = 0;
                                int place = (int)((float)rand()/RAND_MAX*(n/2-i-1));
                                //cout<<place<<endl;
                                for(int l = 0; l < j; l++)
                                        if(nodes[place] == graph[i][l])
                                        {
                                                if(i < n/2-j)
                                                        bad = 1;
                                                else bad = 3;
                                                continue;
                                        }
                                if(bad == 3)
                                        break;
                                if(bad == 1)
                                        continue;
                                b[i][1] = nodes[place];
                                for(l = place; l < n/2-1; l++)
                                        nodes[l] = nodes[l+1];
                        }
                        if(bad == 3)
                        {j--; break;}
                }
                if(bad != 3)
                        for(i = 0; i < n/2; i++)
                                // insert these random edges into the graph
                                // represented by an adjacency list.
                        {
                                graph[i][j] = b[i][1];
                                graph[(int)b[i][1]][j] = i;
                        }
        }
        free(nodes);
        freematrix(n/2, b);
        return graph;
}
```