

## Solution for the Fourth Joint Assignment (part 2)

Danis Alukaev BS19-02

### 2.1. Plot the graph of a time-consumed process and the relation of the predator-prey amount.

Given predator-prey model with initial number of victims  $v_0$ , initial number of killers  $k_0$ , coefficients  $\alpha_1, \beta_1, \alpha_2, \beta_2$  describing the interaction of the two species, time limit  $T$ , and number of the points of approximation  $N$ . Our goal is to plot the graph of a time-consumed process and the relation of the predator-prey amount.

The proposed algorithm (see 2.2 for further details) was evaluated on the following sample inputs:

#### Sample input 1:

$v_0 = 110$   
 $k_0 = 40$   
 $\alpha_1 = 0.4$   
 $\beta_1 = 0.01$   
 $\alpha_2 = 0.3$   
 $\beta_2 = 0.005$   
 $T = 50$   
 $N = 200$

#### Sample input 2:

$v_0 = 6$   
 $k_0 = 6$   
 $\alpha_1 = 0.2$   
 $\beta_1 = 0.025$   
 $\alpha_2 = 0.1$   
 $\beta_2 = 0.02$   
 $T = 200$   
 $N = 1000$

The resultant graphs for these sample inputs shown in Figure 1, 2 (page 2).

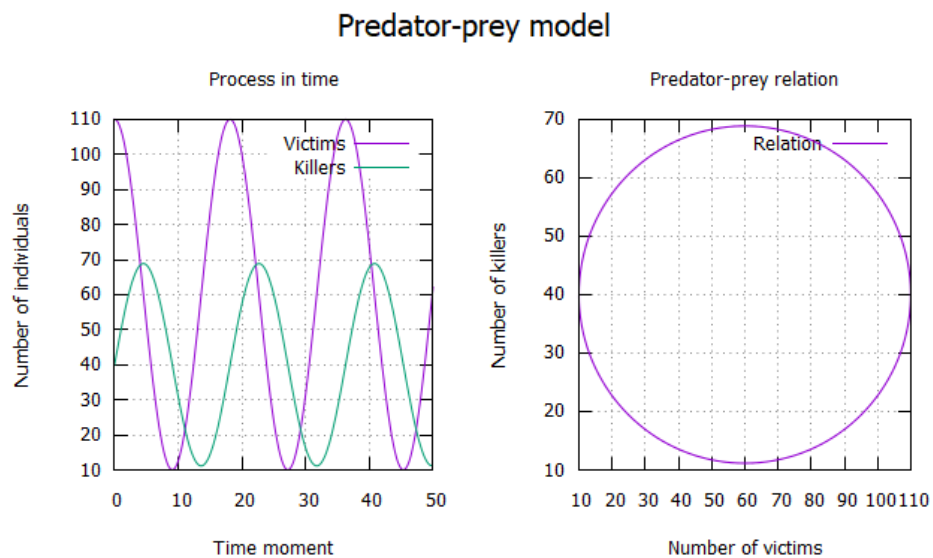


Figure 1: Sample input 1

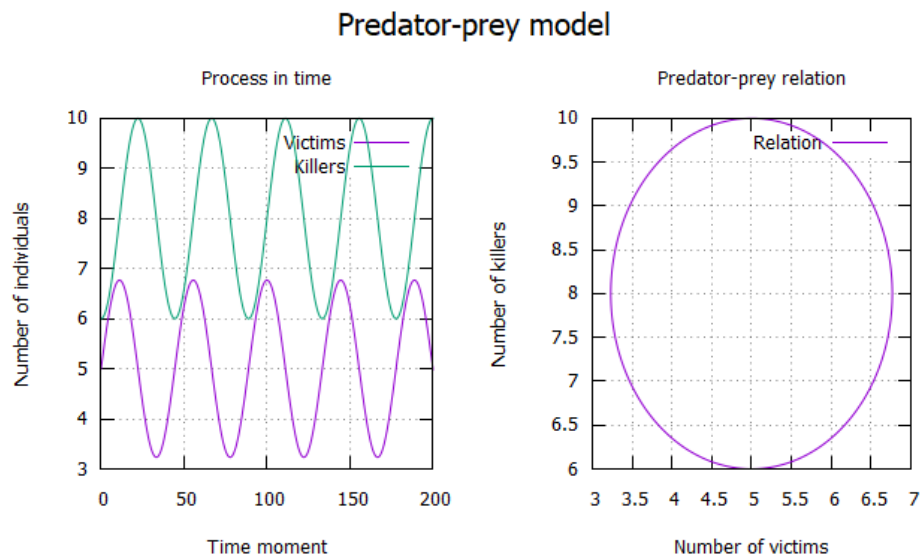


Figure 2: Sample input 2

## 2.2. Implementation of Predator-prey model.

[Online] Available:

[https://github.com/DanisAlukaev/FourthJointAssignment\\_LA\\_II](https://github.com/DanisAlukaev/FourthJointAssignment_LA_II)

The source code is located in file "main.cpp".

[https://github.com/DanisAlukaev/FourthJointAssignment\\_LA\\_II/blob/master/main.cpp](https://github.com/DanisAlukaev/FourthJointAssignment_LA_II/blob/master/main.cpp)

Source code:

```
1 #include <iostream>
2 #include <cmath>
3
4 #define INF (unsigned)!((int)0)
5 #ifdef WIN32
6 #define GNUPLOT_NAME "C:\\gnuplot\\bin\\gnuplot -persist"
7 #endif // WIN32
8
9 using namespace std;
10
11 /**
12  * Fourth Joint Assignment.
13  * #1.
14  * @author Danis Alukaev BS-19-02.
15  */
16
17 /**
18  * Class Matrix.
19  * Represents a rectangular array of numbers arranged in rows
20  * and columns.
21  */
22 class Matrix
23 {
24 public:
25     int n, m; // dimensions of a matrix
26     double **data; // the dynamic array to store elements of a
27     matrix
28
29     /**
30      * Constructor of the class Matrix.
31      * Dynamically allocates memory to store the matrix with the
32      * received number of rows and columns.
33      *
34      * @param rows - the number of rows of a matrix.
35      * @param columns - the number of columns of a matrix.
36      */
37     Matrix(int rows, int columns)
38     {
39         n = rows; // set the number of rows
40         m = columns; // set the number of columns
41     }
42 }
```

```

38     // allocate memory for an array of arrays
39     data = (double **) malloc(sizeof(double*) * n);
40     for(int i = 0; i < n; i++)
41         data[i] = (double*)malloc(sizeof(double) * m);
42 }
43
44 /**
45  * Overloading " >> " operator for a class Matrix
46  */
47 friend istream& operator >> (istream& in, const Matrix&
matrix)
48 {
49     for(int i = 0; i < matrix.n; i++)
50         for(int j = 0; j < matrix.m; j++)
51             in >> matrix.data[i][j]; // read the element
with indexes i, j
52     return in;
53 }
54
55 /**
56  * Overloading " << " operator for a class Matrix
57  */
58 friend ostream& operator << (ostream& out, const Matrix&
matrix)
59 {
60     for(int i = 0; i < matrix.n; i++)
61     {
62         for(int j = 0; j < matrix.m-1; j++)
63             out << matrix.data[i][j] << " ";
64         out << matrix.data[i][matrix.m-1] << "\n"; // print
the element with indexes i, j
65         // use the construction round(someNumber * 100) /
100 to round half towards one
66     }
67     return out;
68 }
69
70 /**
71  * Overloading " = " operator for a class Matrix
72  *
73  * @param other - the matrix to be moved to this instance.
74  * @return *this - this instance of a class Matrix.
75  */
76 Matrix& operator = (Matrix& other)
77 {
78     n = other.n; // set new dimensions
79     m = other.m; // of a matrix
80     data = other.data; // transfer elements to this
instance of a matrix

```

```

81         return *this;
82     }
83
84     /**
85     * Overloading " + " operator for a class Matrix
86     *
87     * @param other - the matrix to be added to this instance.
88     * @return *matrixN - the sum of two matrices.
89     */
90     Matrix& operator + (Matrix& other)
91     {
92         Matrix* matrixN = new Matrix(n, m); // creating new
instance of the class Matrix to store the result
93         for(int i = 0; i < n; i++)
94             for(int j = 0; j < m; j++)
95                 matrixN->data[i][j] = data[i][j] + other.data
[i][j]; // store the result of an addition
96         return *matrixN;
97     }
98
99     /**
100    * Overloading " - " operator for a class Matrix
101    *
102    * @param other - the matrix to be subtracted from this
instance.
103    * @return *matrixN - the difference of two matrices.
104    */
105    Matrix& operator - (Matrix& other)
106    {
107        Matrix* matrixN = new Matrix(n, m); // creating new
instance of the class Matrix to store the result
108        for(int i = 0; i < n; i++)
109            for(int j = 0; j < m; j++)
110                matrixN->data[i][j] = data[i][j] - other.data
[i][j]; // store the result of a subtraction
111        return *matrixN;
112    }
113
114    /**
115    * Overloading " * " operator for a class Matrix
116    *
117    * @param other - the matrix to be multiplied by this
instance.
118    * @return *matrixN - the transposed matrix.
119    */
120    Matrix& operator * (Matrix& other)
121    {
122        Matrix* product = new Matrix(n, other.m); // creating
new instance of the class Matrix to store the result

```

```

123         for(int i = 0; i < n; i++)
124             for(int j = 0; j < other.m; j++)
125                 product -> data[i][j] = 0; // nullify all
positions of a new matrix
126         for(int i = 0; i < n; i++)
127             for(int j = 0; j < other.m; j++)
128                 for(int k = 0; k < m; k++)
129                     product -> data[i][j] += data[i][k] * other
.data[k][j]; // store the result of multiplication
130         return *product;
131     }
132
133     /**
134     * Transposition of the matrix.
135     * Flips a matrix over its principal diagonal.
136     *
137     * @return *matrixN - the transposed matrix.
138     */
139     Matrix& transpose()
140     {
141         Matrix* matrixN = new Matrix(m, n); // creating new
instance of the class Matrix to store the result
142         for(int i = 0; i < m; i++)
143             for(int j = 0; j < n; j++)
144                 matrixN -> data[i][j] = data[j][i]; // store
elements of a particular row in the corresponding column
145         return *matrixN;
146     }
147
148     /**
149     * Destructor of the class Matrix.
150     */
151     ~Matrix()
152     {
153         for(int i = 0; i < n; i++)
154             delete [] data[i];
155         delete [] data;
156     }
157 };
158
159 /**
160 * Class SquareMatrix.
161 * Represents the matrix with the same number of rows and
columns.
162 */
163 class SquareMatrix : public Matrix
164 {
165 public:
166     /**

```

```

167     * Constructor of the class SquareMatrix.
168     * Creates the matrix with the same number of rows and
    columns.
169     *
170     * @param dimension - the dimension of matrix.
171     */
172     SquareMatrix (int dimension) : Matrix(dimension, dimension)
173     {
174         // creating new instance of the class Matrix with the
    received number of rows and columns
175     }
176 };
177
178 /**
179  * Class IdentityMatrix.
180  * Represents the square matrix with ones on the main diagonal
    and zeros elsewhere.
181  */
182 class IdentityMatrix : public SquareMatrix
183 {
184 public:
185     /**
186     * Constructor of the class IdentityMatrix.
187     * Creates the square matrix with ones on the main diagonal
    and zeros elsewhere.
188     *
189     * @param dimension - the dimension of an identity matrix.
190     */
191     IdentityMatrix (int dimension) : SquareMatrix(dimension)
192     {
193         for(int i = 0; i < dimension; i++)
194             for(int j = 0; j < dimension; j++)
195                 i == j ? data[i][j] = 1 : data [i][j] = 0; //
    creating the identity matrix, set the main diagonal
    elements to ones and fill the rest of matrix with zeroes
196     }
197 };
198
199 /**
200  * Class PermutationMatrix.
201  * Represents the square matrix used to exchange two rows with
    received indexes of the matrix.
202  */
203 class PermutationMatrix : public SquareMatrix
204 {
205 public:
206     /**
207     * Constructor of the class PermutationMatrix.
208     * Creates the identity matrix with exchanged columns i1 and

```

```

209     i2.
210     *
211     * @param dimension - the dimension of a permutation matrix.
212     * @param i1 - the first column to be exchanged.
213     * @param i2 - the second column to be exchanged
214     */
215     PermutationMatrix (int dimension, int i1 = 1, int i2 = 1) :
216     SquareMatrix(dimension)
217     {
218         i1--; // since the number of lines of matrix in linear
219         algebra belongs
220         i2--; // to the range [1; +inf], map it to the [0; +inf
221         ]
222         for(int i = 0; i < dimension; i++)
223             for(int j = 0; j < dimension; j++)
224                 i == j ? data[i][j] = 1 : data[i][j] = 0; //
225         creating the identity matrix, set the main diagonal
226         elements to ones and fill the rest of matrix with zeroes
227         data[i2][i2] = 0; // swap corresponding
228         data[i2][i1] = 1; // elements of lines
229         data[i1][i1] = 0; // to make it
230         data[i1][i2] = 1; // permutation matrix
231     }
232 };
233
234 /**
235  * Class EliminationMatrix.
236  * Represents the square matrix used to lead elements with
237  * received indexes of the matrix to zeroes.
238  */
239 class EliminationMatrix : public IdentityMatrix
240 {
241 public:
242     /**
243     * Constructor of the class EliminationMatrix.
244     * Creates the matrix that nullify the corresponding element
245     of the received matrix.
246     *
247     * @param matrix - given matrix, which element [i1, i2]
248     should be zero.
249     * @param i1 - the element's line of the given matrix.
250     * @param i2 - the element's column of the given matrix.
251     */
252     EliminationMatrix (Matrix& matrix, int i1, int i2) :
253     IdentityMatrix(matrix.n)
254     {
255         i1--; // since the number of lines of matrix in linear
256         algebra belongs
257         i2--; // to the range [1; +inf], map it to the [0; +inf

```



```

247     ]
248     // check the potential division by 0
249     try
250     {
251         if (matrix.data[i2][i2] == 0)
252             throw runtime_error("Division by 0");
253         data[i1][i2] = - matrix.data[i1][i2] / matrix.data[
254             i2][i2]; // calculate the coefficient that will nullify the
255             element with received indexes
256     }
257     catch(runtime_error& e)
258     {
259         cout << e.what() << endl;
260     }
261 };
262 /**
263  * Class ScaleMatrix.
264  * Represents the matrix used to lead the diagonal matrix to
265  * the identity matrix.
266  */
267 class ScaleMatrix : public Matrix
268 {
269 public:
270     /**
271     * Constructor of the class ScaleMatrix.
272     * Creates the matrix which principal diagonal elements are
273     reciprocal to corresponding elements of the received matrix
274     .
275     *
276     * @param matrix - the given matrix, which principal
277     diagonal elements should be ones.
278     */
279     ScaleMatrix (Matrix& matrix) : Matrix(matrix.n, matrix.n)
280     {
281         for(int i = 0; i < matrix.n; i++) // treat all
282             for(int j = 0; j < matrix.n; j++) // elements of
283                 the created matrix
284                 data[i][j] = 0; // nullify all elements of a
285                 matrix
286         for(int i = 0; i < matrix.n; i++)
287             data[i][i] = 1 / matrix.data[i][i]; // set elements
288             of the main diagonal to corresponding coefficients
289     }
290 };
291 /**
292  * Class AugmentedMatrix.

```

```

286 * Represents matrix that can be used to perform the same
      elementary row operations on each of the given matrices.
287 * Particularly, in this implementation it applied to find the
      inverse matrix.
288 */
289 class AugmentedMatrix : public Matrix
290 {
291 public:
292     /**
293      * Constructor of the class AugmentedMatrix.
294      * Merges the received and identity matrices by appending
      their columns.
295      *
296      * @param matrix - given matrix to be merged with identity
      matrix.
297      */
298     AugmentedMatrix(Matrix& matrix) : Matrix(matrix.n, 2 *
      matrix.n)
299     {
300         for(int i = 0; i < matrix.n; i++)
301         {
302             for(int j = 0; j < matrix.n; j++) // treat all
      columns from 0 up to n-th
303                 data[i][j] = matrix.data[i][j]; // copy
      elements of received matrix
304             for(int j = matrix.n; j < 2 * matrix.n; j++) //
      treat all columns from n-th up to 2*n-th
305                 i == (j - matrix.n) ? data[i][j] = 1 : data [i
      ][j] = 0; // set the main diagonal elements to ones and
      fill the rest of matrix with zeroes
306         }
307     }
308 };
309
310 /**
311 * Inverses the received matrix using Gaussian Elimination
      approach.
312 *
313 * @param matrix - given matrix to be inversed.
314 * @return inversed - the inversed matrix.
315 */
316 Matrix& getInverse(Matrix& matrix)
317 {
318     Matrix *Augmented = new AugmentedMatrix(matrix); //
      creating an augmented matrix
319     int step = 1, swaps = 0; // the number of steps and
      permutations
320     // nullify elements under the principal diagonal
321     for(int i = 0; i < Augmented->n; i++) // treat all rows of

```

```

322     a matrix
323     {
324         // find the pivot with the maximum absolute value
325         // store its index in the pivotIndex
326         // store its value in the pivotValue
327         int pivotIndex = i;
328         double pivotValue = abs(Augmented->data[i][i]);
329         for(int j = i; j < Augmented->n; j++)
330         {
331             if (pivotValue < abs(Augmented->data[j][i]) && ((
abs(Augmented->data[j][i]) - pivotValue) >= 0.01)) // find
the pivot with maximum absolute value
332             {
333                 pivotIndex = j; // store the index of the found
element
334                 pivotValue = abs(Augmented->data[j][i]); //
store value of the found element
335             }
336             // swap the current line with the found pivot line
337             if(pivotIndex != i)
338             {
339                 Matrix *P = new PermutationMatrix(Augmented->n,
pivotIndex + 1, i + 1); // create the permutation matrix P_
{pivotline+1 i+1} for a current state
340                 *Augmented = *P * (*Augmented); // apply the
permutation matrix
341                 swaps++; // increment the number of permutations
342             }
343             for(int j = i + 1; j < Augmented->n; j++)
344             {
345                 Matrix *E = new EliminationMatrix(*Augmented, j +
1, i + 1); // create the elimination matrix E_{j+1 i+1} for
a current state
346                 *Augmented = *E * (*Augmented); // apply the
elimination matrix
347             }
348         }
349         // nullify elements over the principal diagonal
350         for(int i = Augmented->n-1; i >= 0; i--)
351         {
352             for(int j = i - 1; j >= 0; j--)
353             {
354                 Matrix *E = new EliminationMatrix(*Augmented, j +
1, i + 1); // create the elimination matrix E_{j+1 i+1} for
a current state
355                 *Augmented = *E * (*Augmented); // apply the
elimination matrix
356             }

```

```

357     }
358     // the diagonal normalization
359     Matrix *scale = new ScaleMatrix(*Augmented); // create the
scale matrix for the diagonal normalization
360     *Augmented = *scale * (*Augmented); // perform the diagonal
normalization
361     Matrix *inversed = new SquareMatrix(Augmented->n);
362     // move the right part from n-th up to 2*n-th column of the
augmented matrix to a created "inversed" matrix
363     for(int i = 0; i < Augmented->n; i++)
364         for(int j = Augmented->n; j < 2*Augmented->n; j++)
365             inversed -> data[i][j - Augmented->n] = Augmented->
data[i][j];
366     return *inversed; // return the inversed matrix
367 }
368
369 /**
370  * Predator-Prey Model.
371  * Describes the dynamics of a biological system in which two
species interact, one as a predator and the other as prey.
372  * Method computes populations change through time limit T with
quantization resolution N.
373  *
374  * @param v0 - initial number of victims.
375  * @param k0 - initial number of killers.
376  * @param a1 - positive real parameter describing the rate of
increase in the prey population (reproduction coefficient).
377  * @param b1 - positive real parameter describing the rate of
decrease in the prey population (hunting coefficient).
378  * @param a2 - positive real parameter describing the rate of
decrease in the predator population (natural selection
coefficient).
379  * @param b2 - positive real parameter describing the rate of
increase in the predator population (reproduction
coefficient).
380  * @param T - time limit.
381  * @param N - number of the points of approximation.
382  */
383 void modelPredatorPrey(int v0, int k0, double a1, double b1,
double a2, double b2, int T, int N)
384 {
385
386 #ifdef WIN32
387     FILE* pipe = _popen(GNUPLOT_NAME, "w");
388 #endif
389
390     double interval = (double) T / (double) N; // compute time
interval of quantization
391     Matrix* timeMoments = new Matrix(1, N + 1); // array of

```

```

time moments
392 Matrix* victimsPopulation = new Matrix(1, N + 1); // array
of victim population size
393 Matrix* killerPopulation = new Matrix(1, N + 1); // array
of killer population size
394 k0 -= a1 / b1; // equilibrium deduction
395 v0 -= a2 / b2; // equilibrium deduction
396 timeMoments -> data[0][0] = 0; // set first time moment to
a 0.00
397 for(int i = 1; i < N + 1; i++)
398     // set the rest of time moments according to the
quantization resolution
399     timeMoments -> data[0][i] = timeMoments -> data[0][i -
1] + interval;
400 cout << "t:\n" << *timeMoments; // output array of time
moments
401 for(int i = 0; i < N + 1; i++)
402 {
403     double t = timeMoments -> data[0][i]; // get a time
moment
404     // calculate victim population size at the time moment
t
405     victimsPopulation -> data[0][i] = v0 * cos(sqrt(a1 * a2
) * t) - k0 * sqrt(a2) * b1 / (b2 * sqrt(a1)) * sin(sqrt(a1
* a2) * t) + a2 / b2;
406 }
407 cout << "v:\n" << *victimsPopulation; // output array of
victim population size
408 for(int i = 0; i < N + 1; i++)
409 {
410     double t = timeMoments -> data[0][i]; // get a time
moment
411     // calculate killer population size at the time moment
t
412     killerPopulation -> data[0][i] = v0 * sqrt(a1) * b2 / (
b1 * sqrt(a2)) * sin(sqrt(a1 * a2) * t) + k0 * cos(sqrt(a1
* a2) * t) + a1 / b1;
413 }
414 cout << "k:\n" << *killerPopulation; // output array of
killer population size
415
416 // PLOTTING AUGMENTED CHART
417 if(pipe != NULL)
418 {
419     // chart title
420     fprintf(pipe, "%s\n", "set tics font ',8'");
421     fprintf(pipe, "%s\n", "set multiplot title 'Predator-prey
model' font ',12' layout 1,2");
422     // labels of axis

```

```

423     fprintf(pipe, "%s\n", "set grid\nset xlabel 'Time
moment' font ',8'\nset ylabel 'Number of individuals' font
',8'\n");
424     // Process in time
425     fprintf(pipe, "%s\n", "set title 'Process in time' font
',8'\n");
426     fprintf(pipe, "%s\n", "set key font ',8'\n");
427     fprintf(pipe, "%s\n", "plot '-' using 1:2 title '
Victims' with lines, '-' using 1:2 title 'Killers' with
lines");
428     for(int i = 0; i < N + 1; i++)
429         // scatter plot of prey population size
430         fprintf(pipe, "%f\t%f\n", timeMoments -> data[0][i
], victimsPopulation -> data[0][i]);
431     fprintf(pipe, "%s\n", "e");
432     for(int i = 0; i < N + 1; i++)
433         // scatter plot of predator population size
434         fprintf(pipe, "%f\t%f\n", timeMoments -> data[0][i
], killerPopulation -> data[0][i]);
435     fprintf(pipe, "%s\n", "e");
436     fflush(pipe);
437     // Predator-prey relation
438     fprintf(pipe, "%s\n", "set title 'Predator-prey
relation' ");
439     // labels of axis
440     fprintf(pipe, "%s\n", "set grid\nset xlabel 'Number of
victims' font ',8'\nset ylabel 'Number of killers' font
',8'\n");
441     fprintf(pipe, "%s\n", "plot '-' using 1:2 title '
Relation' with lines");
442     for(int i = 0; i < N + 1; i++)
443         // scatter plot of relation between populations
444         sizes
445         fprintf(pipe, "%f\t%f\n", victimsPopulation -> data
[0][i], killerPopulation -> data[0][i]);
446     fprintf(pipe, "%s\n", "e");
447     fflush(pipe);
448     fprintf(pipe, "%s\n", "unset multiplot");
449 }
450 #ifdef WIN32
451     _pclose(pipe);
452 #endif // WIN32
453 }
454 }
455
456 int main()
457 {
458     cout.setf(ios::fixed); // set the decimal precision of

```

```

459     cout.precision(2);          // output values
460     int v0, k0, T, N; // the initial number of victims, the
                          initial number of killers, the time limit, the number of
                          the points of approximation resp.
461     double a1, b1, a2, b2; // coefficients to compose the rate
                          of change of victim and killer populations
462     cin >> v0 >> k0 >> a1 >> b1 >> a2 >> b2 >> T >> N;
463     modelPredatorPrey(v0, k0, a1, b1, a2, b2, T, N);
464 }

```

Listing 1: Implementation of Predator-prey model