

1 Parts A & C

2 Part E

3 Part F

../IterMap.java

```
1 // IterMap.java
2 // Study the convergence of r.k in the logistic map
3
4 import javax.swing.*;
5 import P251.*;
6
7 public class IterMap extends P251Applet {
8
9     /**
10      * ***** VARIABLES *****
11      */
12
13     double tol = 1e-8; // tolerance for N-R convergence
14     int iterMax = 1000000; // max number of iterations in N-R
15     double epsilon = 1e-6; // delta for use in numerical deriv
16
17     int nR; // number of rsk to look for
18     double delta; // scaling parameter for r
19     double alpha; // scaling parameter for y
20
21     double [] roots; // discovered roots
22     double [] y; // value of iterf half way through a 2^k cycle
23
24     double xMax = .5; // x where maximum occurs for logistic map
25
26     JPanel ip1;
27
28     /**
29      * ***** METHODS *****
30      */
31
32     /** Custom Math Functions */
33
34     double f (double x, double r) {
35         // logistic map
36         return r * x * (1 - x);
37     }
38
39     double iterf (double x, double r, int n) {
40         // return n-th iterate of f
41         if (n>1) return f(iterf(x,r,n-1), r);
42         else return f(x,r);
43     }
44
45     double F (double r, int k) {
46         // function used for optimization of r when x=xMax
47         //
48         double ans = iterf(.5, r, (int) Math.pow(2,k)) - xMax;
49         for (int i=0; i<nR; i++){
50             ans /= (r-roots[i]);
51         }
52
53         return ans;
54     }
55 }
```

```
56
57 double dFdr (double r, int k){
58 // take the derivative of F WRT r
59 return (F(r+epsilon, k) - F(r, k)) / epsilon;
60 }
61
62 double rootR (double rg, int k){
63 // use Newton's method to find r_{si}
64
65 double r = rg;
66 double delta;
67 int i = 0;
68
69 do {
70 // find derivative towards zero
71 delta = -F(r,k) / dFdr(r,k);
72 // update r
73 r += delta;
74
75 if (i>=iterMax) break;
76 i++;
77 } while(Math.abs(delta)>tol);
78
79 // if we timed out, then return NaN
80 if (i==iterMax) {
81 r = Double.NaN;
82 System.out.println("Tried too hard to find rootR");
83 }
84
85
86 return r;
87 }
88
89 /***** P251Applet Methods *****/
90 public void fillPanels() {
91 // define the panels for human interaction
92 ip1 = new inputPanel();
93
94 ip1.addField("nR", 10);
95 addPanel(ip1);
96 initValues();
97
98 }
99
100 public void initValues() {
101 // set up initial values
102
103 nR = 10;
104 roots = new double[nR];
105 y = new double[nR];
106 // set roots to 1 so as not to cause overflow on divide
107 for (int i=0; i<nR; i++) roots[i] = 1;
108
109 }
110
111 public void readValues() {
```

```
112 // read input panel values
113 nR = (int) ip1.getValue(0);
114 roots = new double[nR];
115 y = new double[nR];
116 // set roots to 1 so as not to cause overflow on divide
117 for (int i=0; i<nR; i++) roots[i] = 1;
118
119 }
120
121 public void compute() {
122
123     double rg; // r guess for root finding
124
125     // solved for first two analytically
126     roots[0] = 2;
127     roots[1] = 1 + Math.sqrt(5);
128
129     System.out.println("\nk\ttr_sk\ttd\ty\ta");
130
131     // solve details of k=0,1 cases outside of the loop
132     int k = 0;
133     y[k] = iterf(.5, roots[k], (int) Math.pow(2,k-1));
134     System.out.println(String.format("%02d\t%4.4f\t--\t%4.4f\t--", k+1, roots[k],
135                                     y[k]));
136
137     k = 1;
138     y[k] = iterf(.5, roots[k], (int) Math.pow(2,k-1));
139     System.out.println(String.format("%02d\t%4.4f\t--\t%4.4f\t--", k+1, roots[k],
140                                     y[k]));
141
142     // for each k, figure out r_sk, delta, y, alpha
143     for (k=2; k<nR; k++) {
144         rg = roots[k-1] + .1 * (roots[k-1] - roots[k-2]);
145         roots[k] = rootR(rg, k);
146         delta = (roots[k-1] - roots[k-2]) / (roots[k]-roots[k-1]);
147         y[k] = iterf(.5, roots[k], (int) Math.pow(2,k-1));
148         alpha = - (y[k-1] - y[k-2]) / (y[k] - y[k-1]);
149         System.out.println(String.format("%02d\t%4.4f\t%4.4f\t%4.4f", k+1,
150                                         roots[k], delta, y[k], alpha));
151
152         if (Thread.interrupted()) return;
153     }
154 }
155 }
156
157 }
```

../IterMap2.java

```
1 // IterMap2.java
2 // Study the convergence of r_k in an iterative map
3
4 import javax.swing.*;
5 import P251.*;
6
7 public class IterMap2 extends P251Applet {
8
9     /**
10      * ***** VARIABLES *****
11      */
12
13     double tol = 1e-8; // tolerance for N-R convergence
14     int iterMax = 1000000; // max number of iterations in N-R
15     double epsilon = 1e-6; // delta for use in numerical deriv
16
17     int nR; // number of rsk to look for
18     double delta; // scaling parameter for r
19     double alpha; // scaling parameter for y
20
21     double [] roots; // discovered roots
22     double [] y; // value of iterf half way through a 2^k cycle
23
24     double X0 = .8; // initialize X0 for iteration at startup
25
26     double xMax = .8; // x where maximum occurs for f
27     double rMin = 2.6; // just before the first bifurcation
28     double rMax = 3.5; // once f leaves the range of interest
29     double rStep; // how finely to look at r in the bifurcation map
30
31     inputPanel ip1;
32     graphPanel gp1;
33
34
35     /**
36      * ***** METHODS *****
37      */
38
39     /** Custom Math Functions */
40
41     double f (double x, double r) {
42         // Gr (RP3.8)
43         return r * x*x * Math.sqrt(1 - x);
44     }
45
46     double iterf (double x, double r, int n) {
47         // return n-th iterate of f
48         if (n>1) return f(iterf(x,r,n-1), r);
49         else return f(x,r);
50     }
51
52     double F (double r, int k) {
53         // function used for optimization of r when x=xMax
54         //
55         double ans = iterf(xMax, r, (int) Math.pow(2,k)) - xMax;
```

```
56     for (int i=0; i<nR; i++){
57         ans /= (r-roots[i]);
58     }
59     return ans;
60 }
61
62 double dFdr (double r, int k){
63     // take the derivative of F WRT r
64     return (F(r+epsilon, k) - F(r, k)) / epsilon;
65 }
66
67 double rootR (double rg, int k){
68     // use Newton's method to find r_{si}
69
70     double r = rg;
71     double delta;
72     int i = 0;
73
74     do {
75         // find derivative towards zero
76         delta = -F(r,k) / dFdr(r,k);
77         // update r
78         r += delta;
79
80         if (i>=iterMax) break;
81         i++;
82     } while(Math.abs(delta)>tol);
83
84     // if we timed out, then return NaN
85     if (i==iterMax) {
86         r = Double.NaN;
87         System.out.println("Tried too hard to find rootR");
88     }
89
90
91     return r;
92 }
93
94 double [] getLastValues(double r, int num, int nit) {
95     // get the last num values of f
96     // after iterating nit times
97     double [] output = new double[num];
98     double X = X0;
99     for (int i=0; i<nit; i++) {
100         X = f(X, r);
101         if (i>nit-num-1) {
102             output[i + num - nit] = X;
103         }
104     }
105     return output;
106 }
107
108
109 /***** Plotting Methods *****/
110 void plotBifurcations(double rStep, graphPanel gp, int nKeep, int nIter){
111     gp.clear();
```

```
112 double [] RR = new double[nKeep];
113 double [] XX = new double[nKeep];
114
115 double R = rMin;
116 while (R<=rMax) {
117     XX = getLastValues(R, nKeep, nIter);
118     for (int i = 0; i<nKeep; i++) RR[i] = R;
119
120     gp.addData(RR, XX, "bifurcation");
121     R += rStep;
122     if (Thread.interrupted()) return;
123 }
124 }
125
126 /***** P251Applet Methods *****/
127 public void fillPanels() {
128     // define the panels for human interaction
129     ip1 = new inputPanel();
130
131     ip1.addField("nR", 10);
132     ip1.addField("rStep", .001);
133     addPanel(ip1);
134     initValues();
135
136     gp1 = new graphPanel(600, 300, false);
137
138     gp1.setXLabel("R");
139     gp1.setYLabel("f(X)");
140     gp1.setTitle("Bifurcation Diagram");
141     addPanel(gp1);
142
143 }
144
145 public void initValues() {
146     // set up initial values
147
148     nR = 10;
149     rStep = .001;
150     roots = new double[nR];
151     y = new double[nR];
152     // set roots to 1 so as not to cause overflow on divide
153     for (int i=0; i<nR; i++) roots[i] = 1;
154
155 }
156
157 public void readValues() {
158     // read input panel values
159     nR = (int) ip1.getValue(0);
160     rStep = ip1.getValue(1);
161     roots = new double[nR];
162     y = new double[nR];
163     // set roots to 1 so as not to cause overflow on divide
164     for (int i=0; i<nR; i++) roots[i] = 1;
165
166 }
167
```

```
168 public void compute() {
169
170 double rg; // r guess for root finding
171
172 // solved for first two analytically
173 roots[0] = 2.79508;
174 roots[1] = 3.15783;
175
176 System.out.println("\nk\td\ta");
177
178 // solve details of k=0,1 cases outside of the loop
179 int k = 0;
180 y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
181 System.out.println(String.format("%02d\t--\t—", k+1));
182
183 k = 1;
184 y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
185 System.out.println(String.format("%02d\t--\t—", k+1));
186
187 // for each k, figure out r_sk, delta, y, alpha
188 for (k=2; k<nR; k++) {
189     rg = roots[k-1] + .1 * (roots[k-1] - roots[k-2]);
190     roots[k] = rootR(rg, k);
191     delta = (roots[k-1] - roots[k-2]) / (roots[k]-roots[k-1]);
192     y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
193     alpha = - (y[k-1] - y[k-2]) / (y[k] - y[k-1]);
194     System.out.println(String.format("%02d\t%4.4f\t%4.4f", k+1, delta, alpha));
195
196     if (Thread.interrupted()) return;
197 }
198
199 plotBifurcations(rStep, gpl, 100, 1000);
200
201
202
203
204
205 }
206
207 }
```


../IterMap3.java

```
1 // IterMap3.java
2 // Study the convergence of r_k in the cusped map
3
4 import javax.swing.*;
5 import P251.*;
6
7 public class IterMap3 extends P251Applet {
8
9     /**
10      * ***** VARIABLES *****
11      */
12
13     double tol = 1e-8; // tolerance for N-R convergence
14     int iterMax = 1000000; // max number of iterations in N-R
15     double epsilon = 1e-6; // delta for use in numerical deriv
16
17     int nR; // number of rsk to look for
18     double delta; // scaling parameter for r
19     double alpha; // scaling parameter for y
20
21     double [] roots; // discovered roots
22     double [] y; // value of iterf half way through a 2^k cycle
23
24     double X0 = .5; // initialize X0 for iteration at startup
25
26     double xMax = .5; // x where maximum occurs for f
27     double rMin = .9; // just before the first bifurcation
28     double rMax = 2.82; // once f leaves the range of interest
29     double rStep; // how finely to look at r in the bifurcation map
30
31     inputPanel ip1;
32     graphPanel gp1;
33
34
35     /**
36      * ***** METHODS *****
37      */
38
39     /** Custom Math Functions */
40
41     double f (double x, double r) {
42         // cusp map
43         return r * ( Math.pow(.5, 1.5) - Math.pow(Math.abs(x-.5), 1.5) );
44     }
45
46     double iterf (double x, double r, int n) {
47         // return n-th iterate of f
48         if (n>1) return f(iterf(x,r,n-1), r);
49         else return f(x,r);
50     }
51
52     double F (double r, int k) {
53         // function used for optimization of r when x=xMax
54         //
55         double ans = iterf(xMax, r, (int) Math.pow(2,k)) - xMax;
```

```
56     for (int i=0; i<nR; i++){
57         ans /= (r-roots[i]);
58     }
59     return ans;
60 }
61
62 double dFdr (double r, int k){
63     // take the derivative of F WRT r
64     return (F(r+epsilon, k) - F(r, k)) / epsilon;
65 }
66
67 double rootR (double rg, int k){
68     // use Newton's method to find r_{si}
69
70     double r = rg;
71     double delta;
72     int i = 0;
73
74     do {
75         // find derivative towards zero
76         delta = -F(r,k) / dFdr(r,k);
77         // update r
78         r += delta;
79
80         if (i>=iterMax) break;
81         i++;
82     } while(Math.abs(delta)>tol);
83
84     // if we timed out, then return NaN
85     if (i==iterMax) {
86         r = Double.NaN;
87         System.out.println("Tried too hard to find rootR");
88     }
89
90
91     return r;
92 }
93
94 double [] getLastValues(double r, int num, int nit) {
95     // get the last num values of f
96     // after iterating nit times
97     double [] output = new double[num];
98     double X = X0;
99     for (int i=0; i<nit; i++) {
100         X = f(X, r);
101         if (i>nit-num-1) {
102             output[i + num - nit] = X;
103         }
104     }
105     return output;
106 }
107
108
109 /***** Plotting Methods *****/
110 void plotBifurcations(double rStep, graphPanel gp, int nKeep, int nIter){
111     gp.clear();
```

```
112 double [] RR = new double[nKeep];
113 double [] XX = new double[nKeep];
114
115 double R = rMin;
116 while (R<=rMax) {
117     XX = getLastValues(R, nKeep, nIter);
118     for (int i = 0; i<nKeep; i++) RR[i] = R;
119
120     gp.addData(RR, XX, "bifurcation");
121     R += rStep;
122     if (Thread.interrupted()) return;
123 }
124 }
125
126 /***** P251Applet Methods *****/
127 public void fillPanels() {
128     // define the panels for human interaction
129     ip1 = new inputPanel();
130
131     ip1.addField("nR", 10);
132     ip1.addField("rStep", .001);
133     addPanel(ip1);
134     initValues();
135
136     gp1 = new graphPanel(600, 300, false);
137
138     gp1.setXLabel("R");
139     gp1.setYLabel("f(X)");
140     gp1.setTitle("Bifurcation Diagram");
141     addPanel(gp1);
142
143 }
144
145 public void initValues() {
146     // set up initial values
147
148     nR = 10;
149     rStep = .001;
150     roots = new double[nR];
151     y = new double[nR];
152     // set roots to 1 so as not to cause overflow on divide
153     for (int i=0; i<nR; i++) roots[i] = 1;
154
155 }
156
157 public void readValues() {
158     // read input panel values
159     nR = (int) ip1.getValue(0);
160     rStep = ip1.getValue(1);
161     roots = new double[nR];
162     y = new double[nR];
163     // set roots to 1 so as not to cause overflow on divide
164     for (int i=0; i<nR; i++) roots[i] = 1;
165
166 }
167
```

```
168 public void compute() {
169
170 double rg; // r guess for root finding
171
172 // solved for first two analytically
173 roots[0] = Math.pow(2, .5);
174
175 System.out.println("\nk\td\ta");
176
177 // solve details of k=0,1 cases outside of the loop
178 int k = 0;
179 y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
180 System.out.println(String.format("%02d\t--\t—", k+1));
181
182 k = 1;
183 rg = roots[k-1] * 1.1;
184 roots[k] = rootR(rg, k);
185 y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
186 System.out.println(String.format("%02d\t--\t—", k+1));
187
188 // for each k, figure out r_sk, delta, y, alpha
189 for (k=2; k<nR; k++) {
190     rg = roots[k-1] + .1 * (roots[k-1] - roots[k-2]);
191     roots[k] = rootR(rg, k);
192     delta = (roots[k-1] - roots[k-2]) / (roots[k]-roots[k-1]);
193     y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
194     alpha = - (y[k-1] - y[k-2]) / (y[k] - y[k-1]);
195     System.out.println(String.format("%02d\t%4.4f\t%4.4f", k+1, delta, alpha));
196
197     if (Thread.interrupted()) return;
198 }
199
200 plotBifurcations(rStep, gpl, 100, 1000);
201
202
203
204
205 }
206
207
208 }
```