1 Logistic Map

To begin understand the scaling properties in period-doubling systems, I looked at the familiar iterative map defined by

$$f_r(x) = rx(1-x).$$

As shown in `IterMap.java`, I used Newton's method to find out when $f_r^{(2^k)}(x_m) = x_m$. Finding the $\tilde{r}_k$ that satisfy this condition for each k yields a sequence that must converge to the same value, r_∞, as the r_k which are the bifurcation points of f. To understand how $r_k \to r_\infty$, the ratio δ can be calculated as a scaling parameter. A similar parameter α can also be calculated, which describes the scaling of $f \to f_{r_{\inf}}$ by tracking the parameter. The calculated results are shown in the following table:

k	r_{sk}	δ	y	α
01	2.0000	–	0.5000	–
02	3.2361	–	0.8090	–
03	3.4986	4.7089	0.3836	0.7264
04	3.5546	4.6808	0.5460	2.6199
05	3.5667	4.6630	0.4817	2.5252
06	3.5692	4.6684	0.5073	2.5074
07	3.5698	4.6690	0.4971	2.5038
08	3.5699	4.6692	0.5012	2.5031
09	3.5699	4.6692	0.4995	2.5029
10	3.5699	4.6692	0.5002	2.5029

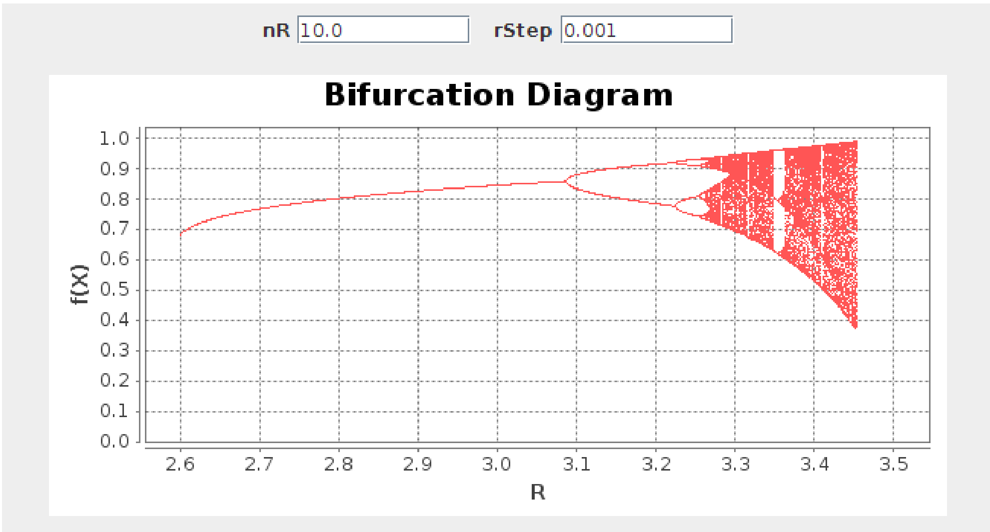
2 Universality

When the function is changed, there is no *a* priori reason to assume that the behavior of r and f will be the same as the logistic map. Using the function

$$G_r(x) = rx^2\sqrt{1-x}$$

I performed the same analysis of δ and α. That these two numbers converge to the same values as for the iterative map should in fact not be surprising. The derivation of these scaling parameters revealed that for any function "like" the logistic map (single quadratic maximum, f(0) = 0) should scale like the logistic map. Thus choice of function became an irrelevant parameter. The table of δ and α are presented here, along with the bifurcation diagram for G_r. The code used to calculate these was `IterMap2.java`.

k	δ	α
01	–	–
02	–	–
03	4.3977	0.6785
04	4.6123	2.2768
05	4.6578	2.6430
06	4.6668	2.4575
07	4.6687	2.5228
08	4.6691	2.4953
09	4.6692	2.5060
10	4.6696	2.5017



3 Renormalization Groups

The emergent universality in the scaling of iterated maps begs the question of how odd our function can be and still behave similarly. To check this, I used a function with a 3/2 power cusp defined by

$$f(x) = r\left((\frac{1}{2})^{3/2} - |x - \frac{1}{2}|^{1}2\right).$$

Performing the same calculations as below yielded the table and bifurcation diagram below (see `IterMap3.java`):

k	δ	α
01	–	–
02	–	–
03	3.8261	0.7856
04	3.7812	3.6168
05	3.7918	3.4416
06	3.7982	3.4019
07	3.7999	3.3921
08	3.8004	3.3896
09	3.8005	3.3889
10	3.8005	3.3887

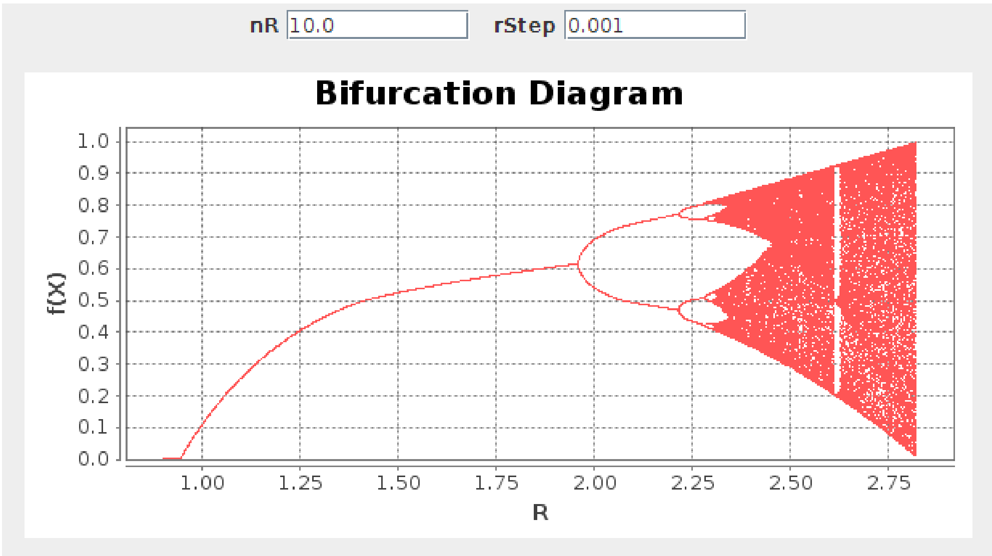
The surprising result here is that the power laws governing approach to r_∞ are have different coefficients than for the logistic map! However, this may not be as crazy as it seems. The derivation of $g(r)$ is the same for all functions f with a single maximum, however, there is an assumption that we used in the estimation of the scaling exponents which is no longer valid. In it, we approximated that $g(z)$ the limiting function of f as $r \to r_\infty$ could be approximated as

$$g(z) \approx A + Bz^2$$

near $z = 0$ because of the quadratic maximum. However, this expansion is not the right expansion for g when f is the cusped function above. In this case, we must approximate

$$g(x) \approx A + B|z|^{3/2}.$$

All other assumptions that led to the Feigenbaum equation were valid, and thus we can use this minor adjustment to reevaluate our expectations for α in this new system.



Feigenbaum can take the form

$$g(z) = -\alpha g(g(-z/\alpha)).$$

With this equation and the approximation, we can write:

$$\begin{aligned} g(z) &= A - B|z|^{3/2} \\ g(g(z)) &= A - B|A - B|z|^{3/2}|^{3/2} \\ &= A - BA^{3/2}|1 - \frac{B}{A}|z|^{3/2} \end{aligned}$$

At this point, we can exploit the fact that z is small, and expand the term inside the absolute values as $(1+\epsilon)^{3/2} \approx 1 + \frac{3}{2}\epsilon$. Doing this and substituting into the Feigenbaum eq. we find that

$$A - B|z|^{3/2} = -\alpha A + \alpha BA^{3/2}(1 - \frac{3B}{2A\alpha^{3/2}}|z|^{3/2})$$

Equating the coefficients of like terms, we find that

$$\begin{aligned} A &= -\alpha A + \alpha BA^{3/2} \\ \Rightarrow BA^{1/2} &= \frac{1+\alpha}{\alpha} \\ B &= \frac{3B^2A^{1/2}}{2\alpha^{1/2}} \\ \Rightarrow 2\alpha^{1/2} &= 3\frac{1+\alpha}{\alpha} \\ \therefore \quad 0 &= 9(1+\alpha)^2 - 4\alpha^3 \end{aligned}$$

Using the previously discovered value of α and `NSolve.java`, I numerically solved for the root of this final equation and calculated an estimate of

$$\alpha \approx 3.65.$$

Comparing this to the value attained by iteration, we find a discrepancy of less than 10%. To improve the agreement, the approximate Feigenbaum equation could include higher order terms.

Regardless of numerical precision, the existence of a second scaling behavior supports the existence of Universality classes, which can broadly describe the behavior of certain functions that meet requirements specific to the class. Here we have seen a new universality class of functions with 3/2 power law cusps. While the period doubling and convergence to r_{inf} behavior are the same as for functions with quadratic maxima, the manner in which they converge differs significantly.

4 Code

../IterMap.java

```java
// IterMap.java
// Study the convergence of r_k in the logistic map

import javax.swing.*;
import P251.*;

public class IterMap extends P251Applet {

    /**********************/
    /****** VARIABLES ******/
    /**********************/

    double tol = 1e-8;       // tolerance for N-R convergence
    int iterMax = 1000000;   // max number of iterations in N-R
    double epsilon = 1e-6;   // delta for use in numerical deriv

    int nR;                  // number of rsk to look for
    double delta;            // scaling parameter for r
    double alpha;            // scaling parameter for y

    double [] roots;         // discovered roots
    double [] y;             // value of iterf half way through a 2^k cycle

    double xMax = .5;        // x where maximum occurs for logistic map

    inputPanel ip1;

    /********************/
    /****** METHODS ******/
    /********************/

    /***** Custom Math Functions *****/

    double f (double x, double r) {
    // logistic map
    return r * x * (1 - x);
    }

    double iterf (double x, double r, int n) {
    // return n-th iterate of f
    if (n>1) return f(iterf(x,r,n-1), r);
    else return f(x,r);
    }

    double F (double r, int k) {
    // function used for optimization of r when x=xMax
        //
    double ans = iterf(.5, r, (int) Math.pow(2,k)) - xMax;
    for (int i=0; i<nR; i++){
        ans /= (r-roots[i]);
    }

```

```
    return ans;

    }

    double dFdr (double r, int k){
    // take the derivative of F WRT r
    return (F(r+epsilon, k) - F(r, k)) / epsilon;
    }

    double rootR (double rg, int k){
    // use Newton's method to find r_{si}

    double r = rg;
    double delta;
    int i = 0;

    do {
        // find derivative towards zero
        delta = -F(r,k) / dFdr(r,k);
        // update r
        r += delta;

        if (i>=iterMax) break;
        i++;
    } while(Math.abs(delta)>tol);

    // if we timed out, then return NaN
    if (i==iterMax) {
        r = Double.NaN;
        System.out.println("Tried too hard to find rootR");
    }


    return r;
    }

    /****** P251Applet Methods ******/
    public void fillPanels() {
    // define the panels for human interaction
    ip1 = new inputPanel();

    ip1.addField("nR", 10);
    addPanel(ip1);
    initValues();

    }

    public void initValues() {
    // set up initial values

    nR = 10;
    roots = new double[nR];
    y = new double[nR];
    // set roots to 1 so as not to cause overflow on divide
    for (int i=0; i<nR; i++) roots[i] = 1;

```

```java
    }

    public void readValues() {
    // read input panel values
    nR = (int) ip1.getValue(0);
    roots = new double[nR];
    y = new double[nR];
    // set roots to 1 so as not to cause overflow on divide
    for (int i=0; i<nR; i++) roots[i] = 1;

    }

    public void compute() {

    double rg; // r guess for root finding

    // solved for first two analytically
    roots[0] = 2;
    roots[1] = 1 + Math.sqrt(5);

    System.out.println("\nk\tr_sk\td\ty\ta");

    // solve details of k=0,1 cases outside of the loop
    int k = 0;
    y[k] = iterf(.5, roots[k], (int) Math.pow(2,k-1));
    System.out.println(String.format("%02d\t%4.4f\t--\t%4.4f\t--", k+1, roots[k], y[k]));

    k = 1;
    y[k] = iterf(.5, roots[k], (int) Math.pow(2,k-1));
    System.out.println(String.format("%02d\t%4.4f\t--\t%4.4f\t--", k+1, roots[k], y[k]));



    // for each k, figure out r_sk, delta, y, alpha
    for (k=2; k<nR; k++) {
        rg = roots[k-1] + .1 * (roots[k-1] - roots[k-2]);
        roots[k] = rootR(rg, k);
        delta = (roots[k-1] - roots[k-2]) / (roots[k]-roots[k-1]);
        y[k] = iterf(.5, roots[k], (int) Math.pow(2,k-1));
        alpha = - (y[k-1] - y[k-2]) / (y[k] - y[k-1]);
        System.out.println(String.format("%02d\t%4.4f\t%4.4f\t%4.4f\t%4.4f", k+1, roots[k], delta, y[k], alpha));

        if (Thread.interrupted()) return;

    }

    }

}
```

../IterMap2.java

```
// IterMap2.java
// Study the convergence of r_k in an iterative map

import javax.swing.*;
import P251.*;

public class IterMap2 extends P251Applet {

    /**********************/
    /****** VARIABLES ******/
    /**********************/

    double tol = 1e-8;        // tolerance for N-R convergence
    int iterMax = 1000000;    // max number of iterations in N-R
    double epsilon = 1e-6;    // delta for use in numerical deriv

    int nR;               // number of rsk to look for
    double delta;         // scaling parameter for r
    double alpha;         // scaling parameter for y

    double [] roots;      // discovered roots
    double [] y;          // value of iterf half way through a 2^k cycle

    double X0 = .8;     // initialize X0 for iteration at startup

    double xMax = .8;   // x where maximum occurs for f
    double rMin = 2.6;  // just before the first bifurcation
    double rMax = 3.5;  // once f leaves the range of interest
    double rStep;       // how finely to look at r in the bifurcation map

    inputPanel ip1;
    graphPanel gp1;


    /********************/
    /****** METHODS ******/
    /********************/

    /***** Custom Math Functions *****/

    double f (double x, double r) {
    // Gr (RP3.8)
    return r * x*x * Math.sqrt(1 - x);
    }

    double iterf (double x, double r, int n) {
    // return n-th iterate of f
    if (n>1) return f(iterf(x,r,n-1), r);
    else return f(x,r);
    }

    double F (double r, int k) {
    // function used for optimization of r when x=xMax

    double ans = iterf(xMax, r, (int) Math.pow(2,k)) - xMax;
```

```
    for (int i=0; i<nR; i++){
        ans /= (r-roots[i]);
    }
    return ans;
    }

    double dFdr (double r, int k){
    // take the derivative of F WRT r
    return (F(r+epsilon, k) - F(r, k)) / epsilon;
    }

    double rootR (double rg, int k){
    // use Newton's method to find r_{si}

    double r = rg;
    double delta;
    int i = 0;

    do {
        // find derivative towards zero
        delta = -F(r,k) / dFdr(r,k);
        // update r
        r += delta;

        if (i>=iterMax) break;
        i++;
    } while(Math.abs(delta)>tol);

    // if we timed out, then return NaN
    if (i==iterMax) {
        r = Double.NaN;
        System.out.println("Tried too hard to find rootR");
    }


    return r;
    }

    double [] getLastValues(double r, int num, int nit) {
    // get the last num values of f
    // after iterating nit times
    double [] output = new double[num];
    double X = X0;
    for (int i=0; i<nit; i++) {
        X = f(X, r);
        if (i>nit-num-1) {
        output[i + num - nit] = X;
        }
    }
    return output;
    }


    /****** Plotting Methods ******/
    void plotBifurcations(double rStep, graphPanel gp, int nKeep, int nIter){
    gp.clear();
```

```
    double [] RR = new double[nKeep];
    double [] XX = new double[nKeep];

    double R = rMin;
    while (R<=rMax) {
        XX = getLastValues(R, nKeep, nIter);
        for (int i = 0; i<nKeep; i++) RR[i] = R;

        gp.addData(RR, XX, "bifurcation");
        R += rStep;
        if (Thread.interrupted()) return;
    }
    }

    /****** P251Applet Methods ******/
    public void fillPanels() {
    // define the panels for human interaction
    ip1 = new inputPanel();

    ip1.addField("nR", 10);
    ip1.addField("rStep", .001);
    addPanel(ip1);
    initValues();

    gp1 = new graphPanel(600, 300, false);

    gp1.setXLabel("R");
    gp1.setYLabel("f(X)");
    gp1.setTitle("Bifurcation Diagram");
    addPanel(gp1);

    }

    public void initValues() {
    // set up initial values

    nR = 10;
    rStep = .001;
    roots = new double[nR];
    y = new double[nR];
    // set roots to 1 so as not to cause overflow on divide
    for (int i=0; i<nR; i++) roots[i] = 1;

    }

    public void readValues() {
    // read input panel values
    nR = (int) ip1.getValue(0);
    rStep = ip1.getValue(1);
    roots = new double[nR];
    y = new double[nR];
    // set roots to 1 so as not to cause overflow on divide
    for (int i=0; i<nR; i++) roots[i] = 1;

    }

```

```
    public void compute() {

    double rg; // r guess for root finding

    // solved for first two analytically
    roots[0] = 2.79508;
    roots[1] = 3.15783;

    System.out.println("\nk\td\ta");

    // solve details of k=0,1 cases outside of the loop
    int k = 0;
    y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
    System.out.println(String.format("%02d\t--\t—", k+1));

    k = 1;
    y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
    System.out.println(String.format("%02d\t--\t—", k+1));

    // for each k, figure out r_sk, delta, y, alpha
    for (k=2; k<nR; k++) {
        rg = roots[k-1] + .1 * (roots[k-1] - roots[k-2]);
        roots[k] = rootR(rg, k);
        delta = (roots[k-1] - roots[k-2]) / (roots[k]-roots[k-1]);
        y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
        alpha = - (y[k-1] - y[k-2]) / (y[k] - y[k-1]);
        System.out.println(String.format("%02d\t%4.4f\t%4.4f", k+1, delta, alpha));

        if (Thread.interrupted()) return;

    }

    plotBifurcations(rStep, gp1, 100, 1000);




    }

}
```

../IterMap3.java

```
// IterMap3.java
// Study the convergence of r_k in the cusped map

import javax.swing.*;
import P251.*;

public class IterMap3 extends P251Applet {

    /**********************/
    /****** VARIABLES ******/
    /**********************/

    double tol = 1e-8;        // tolerance for N-R convergence
    int iterMax = 1000000;    // max number of iterations in N-R
    double epsilon = 1e-6;    // delta for use in numerical deriv

    int nR;                   // number of rsk to look for
    double delta;             // scaling parameter for r
    double alpha;             // scaling parameter for y

    double [] roots;          // discovered roots
    double [] y;              // value of iterf half way through a 2^k cycle

    double X0 = .5;      // initialize X0 for iteration at startup

    double xMax = .5;    // x where maximum occurs for f
    double rMin = .9;    // just before the first bifurcation
    double rMax = 2.82;  // once f leaves the range of interest
    double rStep;        // how finely to look at r in the bifurcation map

    inputPanel ip1;
    graphPanel gp1;


    /*******************/
    /****** METHODS ******/
    /*******************/

    /***** Custom Math Functions *****/

    double f (double x, double r) {
    // cusp map
    return r * ( Math.pow(.5, 1.5) - Math.pow(Math.abs(x-.5), 1.5) );
    }

    double iterf (double x, double r, int n) {
    // return n-th iterate of f
    if (n>1) return f(iterf(x,r,n-1), r);
    else return f(x,r);
    }

    double F (double r, int k) {
    // function used for optimization of r when x=xMax
        //
    double ans = iterf(xMax, r, (int) Math.pow(2,k)) - xMax;
```

```
    for (int i=0; i<nR; i++){
        ans /= (r-roots[i]);
    }
    return ans;
    }

    double dFdr (double r, int k){
    // take the derivative of F WRT r
    return (F(r+epsilon, k) - F(r, k)) / epsilon;
    }

    double rootR (double rg, int k){
    // use Newton's method to find r_{si}

    double r = rg;
    double delta;
    int i = 0;

    do {
        // find derivative towards zero
        delta = -F(r,k) / dFdr(r,k);
        // update r
        r += delta;

        if (i>=iterMax) break;
        i++;
    } while(Math.abs(delta)>tol);

    // if we timed out, then return NaN
    if (i==iterMax) {
        r = Double.NaN;
        System.out.println("Tried too hard to find rootR");
    }


    return r;
    }

    double [] getLastValues(double r, int num, int nit) {
    // get the last num values of f
    // after iterating nit times
    double [] output = new double[num];
    double X = X0;
    for (int i=0; i<nit; i++) {
        X = f(X, r);
        if (i>nit-num-1) {
        output[i + num - nit] = X;
        }
    }
    return output;
    }


    /****** Plotting Methods ******/
    void plotBifurcations(double rStep, graphPanel gp, int nKeep, int nIter){
    gp.clear();
```

```
    double [] RR = new double[nKeep];
    double [] XX = new double[nKeep];

    double R = rMin;
    while (R<=rMax) {
        XX = getLastValues(R, nKeep, nIter);
        for (int i = 0; i<nKeep; i++) RR[i] = R;

        gp.addData(RR, XX, "bifurcation");
        R += rStep;
        if (Thread.interrupted()) return;
    }
    }

    /****** P251Applet Methods ******/
    public void fillPanels() {
    // define the panels for human interaction
    ip1 = new inputPanel();

    ip1.addField("nR", 10);
    ip1.addField("rStep", .001);
    addPanel(ip1);
    initValues();

    gp1 = new graphPanel(600, 300, false);

    gp1.setXLabel("R");
    gp1.setYLabel("f(X)");
    gp1.setTitle("Bifurcation Diagram");
    addPanel(gp1);

    }

    public void initValues() {
    // set up initial values

    nR = 10;
    rStep = .001;
    roots = new double[nR];
    y = new double[nR];
    // set roots to 1 so as not to cause overflow on divide
    for (int i=0; i<nR; i++) roots[i] = 1;

    }

    public void readValues() {
    // read input panel values
    nR = (int) ip1.getValue(0);
    rStep = ip1.getValue(1);
    roots = new double[nR];
    y = new double[nR];
    // set roots to 1 so as not to cause overflow on divide
    for (int i=0; i<nR; i++) roots[i] = 1;

    }

```

```
    public void compute() {

    double rg; // r guess for root finding

    // solved for first analytically
    roots[0] = Math.pow(2, .5);

    System.out.println("\nk\td\ta");

    // solve details of k=0,1 cases outside of the loop
    int k = 0;
    y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
    System.out.println(String.format("%02d\t--\t--", k+1));

    k = 1;
    rg = roots[k-1] * 1.1;
    roots[k] = rootR(rg, k);
    y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
    System.out.println(String.format("%02d\t--\t--", k+1));

    // for each k, figure out r_sk, delta, y, alpha
    for (k=2; k<nR; k++) {
        rg = roots[k-1] + .1 * (roots[k-1] - roots[k-2]);
        roots[k] = rootR(rg, k);
        delta = (roots[k-1] - roots[k-2]) / (roots[k]-roots[k-1]);
        y[k] = iterf(xMax, roots[k], (int) Math.pow(2,k-1));
        alpha = - (y[k-1] - y[k-2]) / (y[k] - y[k-1]);
        System.out.println(String.format("%02d\t%4.4f\t%4.4f", k+1, delta, alpha));

        if (Thread.interrupted()) return;

    }

    plotBifurcations(rStep, gp1, 100, 1000);




    }

}
```

../NSolve.java

```java
// NSolve.java
// Newton's Method Numerical Solver
// for one dimensional functions, looks for single root at a time
// just change f and xguess and recompile for different functions

import javax.swing.*;
import P251.*;

public class NSolve extends P251Applet {

    // Global Variables

    double defaultTolerance = 1e-9;  // tolerance for N-R convergence
    double derivDelta = 1e-6;        // delta for use in numerical deriv
    int iterMax = 1000000;           // max number of iterations in N-R

    double xg;  // x guess for solver

    private inputPanel ip1;

    // functions


    /*************** Customize Function Here **/

    double F(double x) {
    // function to solve for root

    return 4 * Math.pow(x,3) - 9 * ( Math.pow(x,2) + 2 * x + 1 );

    }

    /****************************************/


    double dFdx (double x) {
    // numerical first derivative of F
    return (F(x+derivDelta) - F(x)) / derivDelta;
    }

    double root (double x0) {
    // Newton's method for root finding
    double x = x0;
    double delta;
    int i = 0;

    do {
        delta = - F(x) / dFdx(x);
        x += delta;

        if (i>=iterMax) break;
        i++;
    } while (Math.abs(delta)>defaultTolerance);

    if (i==iterMax) x = Double.NaN;
```

```java

    return x;
    }



    // Applet Methods

    public void fillPanels() {
    ip1 = new inputPanel();
    ip1.addField("xg", 0);

    addPanel(ip1);

    }

    public void initValues() {
    xg = 0;
    }

    public void readValues() {
    xg = ip1.getValue(0);
    }

    public void compute() {
    double r = root(xg);
    System.out.println("\nroot: " + r);
    }


}
```