ANALYSIS OF METHODS OF NUMERICAL ANALYSIS

JOSEPH R. HUNT

1. Introduction

This project was prepared with LaTeX for my MATH 153 Honors Contract during the fall semester of 2009. It is supplementary material for the Java programs I wrote to explore the methods of numerical analysis described in Chapter 2 of *Numerical Analysis* by Burden Faires [1].

2. Bisection

The bisection method is a root-finding algorithm. It is an iterative algorithm that works by repeatedly bisecting an interval then selecting the subinterval in which a root is suspected to be. It is a very simple method but the cost of that simplicity is it's relative inefficiency.

2.1. **The Algorithm.** The algorithm takes endpoints a,b; tolerance TOL, and a maximum number of iterations N_0, as inputs.

Require: $f(a)$ and $f(b)$ have opposite signs.
  $i = 1$
  while $i \leq N_0$ **do**
    $p = a + (b - a)/2$
    if $f(p) = 0$ or $\frac{(b-a)}{2} < TOL$ **then**
      return p
    end if
    $i = i + 1$
    if $f(a)f(p) > 0$ **then**
      $a = p$
    else
      $b = p$
  end while
  print Method failed to return after N_0 iterations.

2.2. **Analysis.** This algorithm relies on the Intermediate Value Theorem which states that if $f(x)$ is continuous for interval $[a, b]$ and $f(a)$ and $f(b)$ have opposite signs then there exists p in interval $[a, b]$ where $f(p) = 0$. This method will, albeit slowly, always converge towards a solution. There are, however, several problems with this method. For one, there could be any number of roots in the interval (a, b). Secondly, more accurate intermediate approximations of the actual root might be inadvertently discarded. Running the algorithm on a computer also raises several other issues, mostly due to floating point arithmetic. When multiplying $f(a)$ by $f(p)$ there is a chance of an underflow to 0 though it is unlikely because both values converge to 0. The second possible problem arises when when the tolerance is too small of a value to be represented correctly with floats or doubles.

3. Newton's Method

Newton's Method is an iterative root-finding algorithm best known for it's successively better approximations. The method works by taking an initial guess which is near the true root then the function is approximated with it's tangent line (using the derivative). The method is repeated using the x-intercept of the tangent line.

3.1. Algorithm.

This algorithm takes the inputs: initial approximation x_0, tolerance TOL, maximum number of iterations N_0

```
i = 1
while i ≤ N_0 do
   x = x_0 − f(x_0)/f'(x_0)
   if x − x_0 < TOL then
      return x
   end if
   i = i + 1
   x_0 = x
end while
print Method failed to return after N_0 iterations.
```

3.2. Analysis.

The method converges very quickly for most functions given a good first guess and a small interval. The rate of convergence for most functions is at least quadratic near zero, which means the number of correct digits should double every step. Since this function makes use of the tangent line, it can be used to find the maximums and minimums of a function as well since the derivative is 0 at minimums and maximums. There are some difficulties with the method however. First, if the initial guess is too far from the true zero, the method may fail to converge on the correct zero, or even on any zeroes. Secondly, several problems arise from the use of a derivative in the algorithm.

Since derivatives might be computationally and analytically hard to obtain for complex equations, Newton's method may become unwieldy. Obviously if the derivative is 0 the method will not work. Also, if the derivative is very near 0, the tangent line may go past the actual root. If the derivative of the function is not continuous then the method will fail to converge. If the root being sought occurs more than once on the specified interval then it make take many iterations before the algorithm's quadratic convergence speed is obtained. If the derivative is hard to obtain or store, it might be advantageous to use the secant method described in section 4.

Much about the rate of convergence can be found from the derivatives. If the function is continuously differentiable, its derivative is not 0 at the root, and it has a second derivative at 0, then the rate of convergence is quadratic or faster. However, if the second derivative at the root is 0 the rate of convergence is merely quadratic. If the first derivative is 0 then the convergence is linear. In some cases Newton's method will fail to converge if the starting point is not in the interval where the method converges. If this is the case then the bisection method might be used to find a better initial guess. Another problem arises if the starting point causes the method to oscillate from one x intercept to another without ever converging. A simple function where Newton's method fails to converge is $f(x) = \sqrt[3]{x}$. No matter the starting point Newton's method will not converge on the root.

4. Secant Method

The secant method is an iterative root-finding algorithm that uses the roots of secant lines to better approximate the root of a function. Newton's method generally converges faster but the Secant method has the advantage that it does not need to evaluate both $f(x)$ and $f'(x)$ every step. In fact, since the secant method takes one less operation than Newton's method we can perform two steps with the secant method to every one step with Newton's method, which can sometimes be faster.

4.1. **The Algorithm.** This algorithm takes the inputs: initial approximations x_0, x_1, tolerance TOL, and maximum number of iterations N_0

$i = 2$, $q_0 = f(x_0)$, $q_1 = f(x_1)$
while $i \leq N_0$ **do**
  $x = x_1 - q_1(x_1 - x_0)/(q_1 - q_0)$
  if $x - x_1 < TOL$ **then**
    return x
  end if
  $i = i + 1$
  $x_0 = x_1, q_0 = q_1, x_1 = x, q_1 = f(x)$
end while
print Method failed to return after N_0 iterations.

4.2. **Analysis.** This method is very similar to the method of false position with the only difference being the way the successive terms are defined. There are many interesting things to note about this method. For all functions x_0 and x_1 are interchangeable but that doesn't mean the choice doesn't matter. For some functions like $\sin x$, the choice of the guess determines the speed of convergence. The secant method uses the co-ordinates $(x_{i-1}, f(x_{i-1}))$, the values from the previous iteration. This method does have several pitfalls, however. It does not always converge like the bisection method. The function $xe^{-x} = 0$ with guesses $x_0 = 1.5$ and $x_1 = 2.0$ will fail to converge. With x^3 or $\arctan x$ the method would converge quickly, but if the tolerance was set too high the method would continue to oscillate over 0, never reaching the root (but it would make a really cool picture).

5. Conclusions and Reflections

In conclusion, it seems that if calculating derivatives, either numerically or analytically, is not computationally intensive, Newton's method should have effective results for the majority of the applications. A very good algorithm would have the secant method, Newton's method, and the bisection method all at its disposal and decide which method would converge best for the given iteration. The bisection method can be used as a starter method for many of the other methods to get a good starting approximation.

If I'd had more time on this project I would have liked to improve the GUI. I would also liked to add more methods like the method of false position. I would have liked to include more about the efficiency of the methods as well or offer different versions for execution. Finally, I'd have liked to do more on error analysis.

Appendix A. Abstract Code

AbstractMethod.java

```java
/**
 * @author Randall Hunt
 *
 */
public abstract class AbstractMethod {
    public static final int DEFAULT_MAX_ITERATIONS = 100;
    /** finished? */
    boolean                 finished;
    /** solution? */
    boolean                 foundSolution;
    /** A description of the method */
    public String           DESCRIPTION;
    /** The function f to be used in evaluating */
    public F                f;

    /**
     * @return foundSolution
     */
    public boolean foundSolution() {
        return foundSolution;
    }

    /**
     * @return the finished
     */
    public boolean finished() {
        return finished;
    }

    /**
     * @return the currentI
     */
    public int getCurrentI() {
        return currentI;
    }

    int currentI;

    /** go forward one iteration */
    public abstract void step();

    /** go backwards one iteration */
    public abstract void back();

    /**
     * print step results
```

```
     *
     * @return the stuff to print for this step
     */
    public abstract String printStep();

    /**
     * @return The description of the method in use.
     */
    public abstract String getDescription();
}
```

F.java

```java
/** All of the functions */
public enum F {
    /** x^3 + 4x^2 -10 */
    Poly {
        @Override
        public double eval(double x) {
            return Math.pow(x, 3.0) + 4 * Math.pow(x, 2.0) - 10;
        }

        @Override
        public double deriv(double x) {
            return 3 * Math.pow(x, 2) + 8 * x;
        }

        @Override
        public double integrate(double x, int c) {
            return (Math.pow(x, 4) / 4) + (4 * Math.pow(x, 3) / 3) - 10 * x + c;
        }

        @Override
        public String getDescription() {
            return "x^3 + 4x^2 -10";
        }
    },
    /** x^2 - 2 */
    x2 {
        @Override
        public double eval(double x) {
            return Math.pow(x, 2.0) - 2;
        }

        @Override
        public double deriv(double x) {
            return 2 * x;
        }

        @Override
        public double integrate(double x, int c) {
            return (Math.pow(x, 3.0) / 3) - 2 * x + c;
        }

        @Override
        public String getDescription() {
            return "x^2-2 \n Use 0 for Newtons method to get horizontal tangent.";
        }
    },
    /** x^3 - 3x + 2 */
    x3 {
```

```java
        @Override
        public double eval(double x) {
            return Math.pow(x, 3.0) - 3 * x + 2;
        }

        @Override
        public double deriv(double x) {
            return 3 * Math.pow(x, 2.0) - 3;
        }

        @Override
        public double integrate(double x, int c) {
            // FIX
            return (Math.pow(x, 4.0) / 4) + c;
        }

        @Override
        public String getDescription() {
            return "x^3-3x+2 \n Double root at 1";
        }
    },
    /** e^x */
    eX {
        @Override
        public double eval(double x) {
            if (x < .5 && x > -.5) {
                return Math.expm1(x) + 1;
            }
            return Math.exp(x);
        }

        @Override
        public double deriv(double x) {
            return eval(x);
        }

        @Override
        public double integrate(double x, int c) {
            return eval(x);
        }

        @Override
        public String getDescription() {
            return "e^x";
        }
    },
    /** 4arctan */
    arctanx {
        @Override
```

```
            public double eval(double x) {
                return 4 * Math.atan(x);
            }

            @Override
            public double deriv(double x) {
                return 4 / (Math.pow(x, 2.0) + 1);
            }

            @Override
            public double integrate(double x, int c) {
                // NEEDS FIXING
                return (x * Math.atan(x) - (.5 * Math.log(Math.pow(x, 2.0) + 1))) + c;
            }

            @Override
            public String getDescription() {
                return "4arctan(x) \n use start = 1.5, and maxIterations = 4 for newtons.";
            }
        },

        /** sin(x) */
        sinx {
            @Override
            public double eval(double x) {
                return Math.sin(x);
            }

            @Override
            public double deriv(double x) {
                return Math.cos(x);
            }

            @Override
            public String getDescription() {
                return "sin(x) \n Newton's function can't decide which root.";
            }

            @Override
            public double integrate(double x, int c) {
                // FIX
                return 0;
            }
        },
        /** cos(x) - x^3 */
        cosx3 {

            @Override
            public double deriv(double x) {
```

```java
            return 0;
        }

        @Override
        public double eval(double x) {
            return Math.cos(x) - Math.pow(x, 3.0);
        }

        @Override
        public String getDescription() {
            return "cos(x) - x^3";
        }

        @Override
        public double integrate(double x, int c) {
            // TODO Auto-generated method stub
            return 0;
        }

    };

    /**
     * /** returns f(x)
     */
    public abstract double eval(double x);

    /** returns the derivative at x */
    public abstract double deriv(double x);

    /** returns the integral at x */
    public abstract double integrate(double x, int c);

    /** A textual description of the function */
    public abstract String getDescription();

    /** Returns the integral with c=0 */
    public double integrate(double x) {
        return integrate(x, 0);
    }

    /**
     * Returns the numerical derivative of the function. More accurate than the
     * analytical version for some numbers near 0.
     */
    public double numDeriv(double x) {
        return (eval(x + BIT) - eval(x - BIT)) / (2 * BIT);
    }

    /** The constant for use with numerical derivation */
```

```
    public static final double BIT = 1e-8;
}
```

Appendix B. Methods

Bisection.java

```java
/**
 * This method will find a solution to f(x) = 0 given the continuous function f
 * on the interval[a, b] where f(a) and f(b) have opposite signs.
 *
 * @author Randall Hunt
 *
 */
public class Bisection extends AbstractMethod {
    /** Endpoint */
    private double a         = 0;
    /** Endpoint */
    private double b         = 2;
    /** Tolerance for error */
    private double tolerance = 1E-3;
    /** Current Guess */
    private double p         = a + (b - a) / 2.0;
    /** Maximum number of iterations */
    private int    max       = 30;

    /**
     * @param a
     * @param b
     * @param tolerance
     * @param max
     * @param function
     */
    public Bisection(double a, double b, double tolerance, int max, F function) {
        this.a = a;
        this.b = b;
        this.p = a + (b - a) / 2.0;
        this.tolerance = tolerance;
        this.max = max;
        this.f = function;
        this.currentI = 0;
        this.finished = false;
        this.DESCRIPTION =
                "This method will find a solution to f(x) = 0 "
                        + "given the continuous function f on the"
                        + " interval [a, b] where f(a) and f(b) have opposite signs.";
    }

    @Override
    public void step() {
        if (currentI < max && finished != true) {
            if (Math.abs(b - a) < (2 * tolerance)) {
                finished = true;
```

```
                foundSolution = true;
                return;
            }
            if (f.eval(p) <= 0)
                a = p;
            else
                b = p;

            p = a + (b - a) / 2;

            ++currentI;
        } else {
            finished = true;
        }
    }

    @Override
    public String getDescription() {
        return DESCRIPTION;
    }

    @Override
    public void back() {
        // TODO Auto-generated method stub
    }

    @SuppressWarnings("boxing")
    @Override
    public String printStep() {
        return String.format("\n%d %12f %12f %12f %12f", currentI, a, b, p, f.eval(p));
    }

    /**
     * @param args
     */
    @SuppressWarnings("boxing")
    public static void main(String args[]) {
        double a = 0, b = 0, tol = 0;
        int max = 0;
        F f = F.Poly;
        if (args.length == 5) {
            a = Double.parseDouble(args[0]);
            b = Double.parseDouble(args[1]);
            tol = Double.parseDouble(args[2]);
            max = Integer.parseInt(args[3]);
            f = F.valueOf(args[4]);
        } else if (args.length == 4) {
            a = Double.parseDouble(args[0]);
            b = Double.parseDouble(args[1]);
```

```java
            tol = Double.parseDouble(args[2]);
            max = Integer.parseInt(args[3]);
        } else {
            printUsageAndExit();
        }
        Bisection bisection = new Bisection(a, b, tol, max, f);
        System.out.println(bisection.getDescription());
        System.out.printf("%s: %10s: %10s: %10s: %10s:\n", "i", "a", "b", "p", "f(p)");
        System.out.printf("------------------------------------------------------------");

        Long startTime = System.currentTimeMillis();
        while (!bisection.finished()) {
            System.out.print(bisection.printStep());
            bisection.step();
        }
        Long endTime = System.currentTimeMillis();
        if (bisection.foundSolution())
            System.out.println("\n\nDone\n 0 found at: " + bisection.getP());
        else
            System.out.println("\n\nUnable to find result in " + bisection.getMax()
                    + " iterations.");

        System.out.println("Execution took: " + (endTime - startTime) + " ms and "
                + bisection.currentI + " iterations");
    }

    public static void printUsageAndExit() {
        System.out.println("Usage: java Bisection left right tolerance iterations <functi
        System.out.println("Optional parameter functions can be:");
        for (F f : F.values())
            System.out.println(f.name() + " == " + f.getDescription());
        System.exit(0);
    }

    /**
     * @return the a
     */
    public double getA() {
        return a;
    }

    /**
     * @param a
     *            the a to set
     */
    public void setA(double a) {
        this.a = a;
    }

```

```
    /**
     * @return the b
     */
    public double getB() {
        return b;
    }

    /**
     * @param b
     *            the b to set
     */
    public void setB(double b) {
        this.b = b;
    }

    /**
     * @return the tolerance
     */
    public double getTolerance() {
        return tolerance;
    }

    /**
     * @param tolerance
     *            the tolerance to set
     */
    public void setTolerance(double tolerance) {
        this.tolerance = tolerance;
    }

    /**
     * @return the p
     */
    public double getP() {
        return p;
    }

    /**
     * @param p
     *            the p to set
     */
    public void setP(double p) {
        this.p = p;
    }

    /**
     * @return the max
     */
    public int getMax() {
```

```
        return max;
    }

    /**
     * @param max
     *            the max to set
     */
    public void setMax(int max) {
        this.max = max;
    }
}
```

Newtons.java

```java
public class Newtons extends AbstractMethod {
    private double start         = 0.5;
    private double tolerance     = 1e-14;
    private int    maxIterations = DEFAULT_MAX_ITERATIONS;
    public F       f             = F.Poly;

    public Newtons(double start, double tolerance, int maxIterations, F f) {
        this.start = start;
        this.tolerance = tolerance;
        this.maxIterations = maxIterations;
        this.currentI = 1;
        this.f = f;
    }

    @Override
    public void back() {
        // TO-DO
    }

    @Override
    public String getDescription() {
        return null;
    }

    @Override
    public String printStep() {
        return String.format("\n%d %12f", currentI, start);
    }

    @Override
    public void step() {
        if (currentI < maxIterations && finished != true) {
            if (Math.abs(f.eval(start)) < tolerance) {
                foundSolution = true;
                finished = true;
                return;
            }
            start = start - f.eval(start) / f.deriv(start);
            currentI++;
        } else
            finished = true;
    }

    public static void main(String args[]) {
        if (args.length != 4) {
            printUsageAndExit();
        }
        Newtons newton =
```

```java
                new Newtons(Double.parseDouble(args[0]), Double.parseDouble(args[1]), Int
                        .parseInt(args[2]), F.valueOf(args[3]));
        Long startTime = System.currentTimeMillis();
        while (!newton.finished()) {
            System.out.print(newton.printStep());
            newton.step();
        }
        Long endTime = System.currentTimeMillis();

        if (newton.foundSolution())
            System.out.println("\n\n Done. \n 0 found at: " + newton.getStart());
        else
            System.out.println("\n\n Unable to find result in " + newton.getMaxIterations

        System.out.println("Execution took: " + (endTime - startTime) + " ms and "
                + newton.getCurrentI() + " iterations");

    }

    public static void printUsageAndExit() {
        System.out.println("Usage: java Newton start tolerance iterations <function>");
        System.out.println("Optional parameter functions can be:");
        for (F f : F.values())
            System.out.println(f.name() + " == " + f.getDescription());
        System.exit(0);
    }

    /**
     * @return the start
     */
    public double getStart() {
        return start;
    }

    /**
     * @param start
     *            the start to set
     */
    public void setStart(double start) {
        this.start = start;
    }

    /**
     * @return the tolerance
     */
    public double getTolerance() {
        return tolerance;
    }

```

```
    /**
     * @param tolerance
     *            the tolerance to set
     */
    public void setTolerance(double tolerance) {
        this.tolerance = tolerance;
    }

    /**
     * @return the maxIterations
     */
    public int getMaxIterations() {
        return maxIterations;
    }

    /**
     * @param maxIterations
     *            the maxIterations to set
     */
    public void setMaxIterations(int maxIterations) {
        this.maxIterations = maxIterations;
    }
}
```

Secant.java

```java
/**
 * @author Randall Hunt
 */
public class Secant extends AbstractMethod {
    private double tolerance;
    private double x0;
    private double x1;
    private double x;
    private double q0;
    private double q1;
    private int    maxIterations;

    public F       f;

    public Secant(double x0, double x1, double tolerance, int maxIterations, F f) {
        this.x0 = x0;
        this.x1 = x1;
        this.tolerance = tolerance;
        this.maxIterations = maxIterations;
        this.currentI = 2;
        this.f = f;
        this.q0 = f.eval(x0);
        this.q1 = f.eval(x1);
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        if (args.length != 5) {
            printUsageAndExit();
        }
        Secant secant =
                new Secant(Double.parseDouble(args[0]), Double.parseDouble(args[1]), Dou
                        .parseDouble(args[2]), Integer.parseInt(args[3]), F.valueOf(args
        Long startTime = System.currentTimeMillis();
        while (!secant.finished()) {
            System.out.println(secant.printStep());
            secant.step();
        }
        Long endTime = System.currentTimeMillis();

        if (secant.foundSolution())
            System.out.println("\n\n Done. \n 0 found at: " + secant.getX1());
        else
            System.out.println("\n\n Unable to find result in " + secant.getMaxIterations

        System.out.println("Execution took: " + (endTime - startTime) + " ms and "
```

```java
                + secant.getCurrentI() + " iterations");
    }

    public static void printUsageAndExit() {
        System.out.println("Usage: java Secant x0 x1 tolerance iterations <function>");
        System.out.println("Optional parameter functions can be:");
        for (F f : F.values())
            System.out.println(f.name() + " == " + f.getDescription());
        System.exit(0);
    }

    @Override
    public void back() {
        // TODO
    }

    @Override
    public String getDescription() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public String printStep() {
        return String.format("\n%d %15f %15f", currentI, x1, f.eval(x1));
    }

    @Override
    public void step() {
        x = x1 - q1 * (x1 - x0) / (q1 - q0);
        if (currentI < maxIterations && finished != true) {
            if (Math.abs(x - x1) < tolerance) {
                foundSolution = true;
                finished = true;
                return;
            }
            currentI++;
            x0 = x1;
            q0 = q1;
            x1 = x;
            q1 = f.eval(x);
        } else
            finished = true;
    }

    /**
     * @return the tolerance
     */
    public double getTolerance() {
```

```
        return tolerance;
    }

    /**
     * @param tolerance
     *            the tolerance to set
     */
    public void setTolerance(double tolerance) {
        this.tolerance = tolerance;
    }

    /**
     * @return the x0
     */
    public double getX0() {
        return x0;
    }

    /**
     * @param x0
     *            the x0 to set
     */
    public void setX0(double x0) {
        this.x0 = x0;
    }

    /**
     * @return the x1
     */
    public double getX1() {
        return x1;
    }

    /**
     * @param x1
     *            the x1 to set
     */
    public void setX1(double x1) {
        this.x1 = x1;
    }

    /**
     * @return the maxIterations
     */
    public int getMaxIterations() {
        return maxIterations;
    }

    /**
```

```
     * @param maxIterations
     *            the maxIterations to set
     */
    public void setMaxIterations(int maxIterations) {
        this.maxIterations = maxIterations;
    }
}
```

References

[1] Faires, Burden. *Numerical Analysis*. Boston, MA: PWS-Kent, 1993.