

# ANALYSIS OF METHODS OF NUMERICAL ANALYSIS

JOSEPH R. HUNT

## 1. INTRODUCTION

This project was prepared with  $\text{\LaTeX}$  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 its 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 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 its successively better approximations. The method works by taking an initial guess which is near the true root then the function is approximated with its 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 ≤ N0 do
  x = x0 - f(x0)/f'(x0)
  if x - x0 < TOL then
    return x
  end if
  i = i + 1
  x0 = x
end while
print Method failed to return after N0 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 may 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

```
1  /**
2   * @author Randall Hunt
3   *
4   */
5  public abstract class AbstractMethod {
6      public static final int DEFAULT_MAX_ITERATIONS = 100;
7      /** finished? */
8      boolean                finished;
9      /** solution? */
10     boolean                foundSolution;
11     /** A description of the method */
12     public String          DESCRIPTION;
13     /** The function  $f$  to be used in evaluating */
14     public F                f;
15
16     /**
17      * @return foundSolution
18      */
19     public boolean foundSolution() {
20         return foundSolution;
21     }
22
23     /**
24      * @return the finished
25      */
26     public boolean finished() {
27         return finished;
28     }
29
30     /**
31      * @return the currentI
32      */
33     public int getCurrentI() {
34         return currentI;
35     }
36
37     int currentI;
38
39     /** go forward one iteration */
40     public abstract void step();
41
42     /** go backwards one iteration */
43     public abstract void back();
44
45     /**
46      * print step results
```

```
47      *
48      * @return the stuff to print for this step
49      */
50  public abstract String printStep();
51
52  /**
53   * @return The description of the method in use.
54   */
55  public abstract String getDescription();
56 }
```

## F.java

```
1  /** All of the functions */
2  public enum F {
3      /** x^3 + 4x^2 - 10 */
4      Poly {
5          @Override
6          public double eval(double x) {
7              return Math.pow(x, 3.0) + 4 * Math.pow(x, 2.0) - 10;
8          }
9
10         @Override
11         public double deriv(double x) {
12             return 3 * Math.pow(x, 2) + 8 * x;
13         }
14
15         @Override
16         public double integrate(double x, int c) {
17             return (Math.pow(x, 4) / 4) + (4 * Math.pow(x, 3) / 3) - 10 * x + c;
18         }
19
20         @Override
21         public String getDescription() {
22             return "x^3_+_4x^2_-10";
23         }
24     },
25     /** x^2 - 2 */
26     x2 {
27         @Override
28         public double eval(double x) {
29             return Math.pow(x, 2.0) - 2;
30         }
31
32         @Override
33         public double deriv(double x) {
34             return 2 * x;
35         }
36
37         @Override
38         public double integrate(double x, int c) {
39             return (Math.pow(x, 3.0) / 3) - 2 * x + c;
40         }
41
42         @Override
43         public String getDescription() {
44             return "x^2-2_\n_Use_0_for_Newtons_method_to_get_horizontal_tangent.";
45         }
46     },
47     /** x^3 - 3x + 2 */
48     x3 {
```

```

49     @Override
50     public double eval(double x) {
51         return Math.pow(x, 3.0) - 3 * x + 2;
52     }
53
54     @Override
55     public double deriv(double x) {
56         return 3 * Math.pow(x, 2.0) - 3;
57     }
58
59     @Override
60     public double integrate(double x, int c) {
61         // FIX
62         return (Math.pow(x, 4.0) / 4) + c;
63     }
64
65     @Override
66     public String getDescription() {
67         return "x^3-3x+2\ndouble root at 1";
68     }
69 },
70 /** e^x */
71 eX {
72     @Override
73     public double eval(double x) {
74         if (x < .5 && x > -.5) {
75             return Math.expm1(x) + 1;
76         }
77         return Math.exp(x);
78     }
79
80     @Override
81     public double deriv(double x) {
82         return eval(x);
83     }
84
85     @Override
86     public double integrate(double x, int c) {
87         return eval(x);
88     }
89
90     @Override
91     public String getDescription() {
92         return "e^x";
93     }
94 },
95 /** 4 arctan */
96 arctanx {
97     @Override

```

```

98     public double eval(double x) {
99         return 4 * Math.atan(x);
100     }
101
102     @Override
103     public double deriv(double x) {
104         return 4 / (Math.pow(x, 2.0) + 1);
105     }
106
107     @Override
108     public double integrate(double x, int c) {
109         // NEEDS FIXING
110         return (x * Math.atan(x) - (.5 * Math.log(Math.pow(x, 2.0) + 1))) + c;
111     }
112
113     @Override
114     public String getDescription() {
115         return "4arctan(x)\nuse_start = 1.5, and_maxIterations = 4 for newtons.";
116     }
117 },
118
119 /** sin(x) */
120 sinx {
121     @Override
122     public double eval(double x) {
123         return Math.sin(x);
124     }
125
126     @Override
127     public double deriv(double x) {
128         return Math.cos(x);
129     }
130
131     @Override
132     public String getDescription() {
133         return "sin(x)\nNewton's function can't decide which root.";
134     }
135
136     @Override
137     public double integrate(double x, int c) {
138         // FIX
139         return 0;
140     }
141 },
142 /** cos(x) - x^3 */
143 cosx3 {
144
145     @Override
146     public double deriv(double x) {

```



```

147         return 0;
148     }
149
150     @Override
151     public double eval(double x) {
152         return Math.cos(x) - Math.pow(x, 3.0);
153     }
154
155     @Override
156     public String getDescription() {
157         return "cos(x) - x^3";
158     }
159
160     @Override
161     public double integrate(double x, int c) {
162         // TODO Auto-generated method stub
163         return 0;
164     }
165
166 };
167
168 /**
169  * /** returns f(x)
170  */
171 public abstract double eval(double x);
172
173 /** returns the derivative at x */
174 public abstract double deriv(double x);
175
176 /** returns the integral at x */
177 public abstract double integrate(double x, int c);
178
179 /** A textual description of the function */
180 public abstract String getDescription();
181
182 /** Returns the integral with c=0 */
183 public double integrate(double x) {
184     return integrate(x, 0);
185 }
186
187 /**
188  * Returns the numerical derivative of the function. More accurate than the
189  * analytical version for some numbers near 0.
190  */
191 public double numDeriv(double x) {
192     return (eval(x + BIT) - eval(x - BIT)) / (2 * BIT);
193 }
194
195 /** The constant for use with numerical derivation */

```

```
196     public static final double BIT = 1e-8;  
197 }
```

## APPENDIX B. METHODS

### Bisection.java

```

1  /**
2   * This method will find a solution to  $f(x) = 0$  given the continuous function  $f$ 
3   * on the interval  $[a, b]$  where  $f(a)$  and  $f(b)$  have opposite signs.
4   *
5   * @author Randall Hunt
6   *
7   */
8  public class Bisection extends AbstractMethod {
9      /** Endpoint */
10     private double a          = 0;
11     /** Endpoint */
12     private double b          = 2;
13     /** Tolerance for error */
14     private double tolerance = 1E-3;
15     /** Current Guess */
16     private double p          = a + (b - a) / 2.0;
17     /** Maximum number of iterations */
18     private int    max        = 30;
19
20     /**
21      * @param a
22      * @param b
23      * @param tolerance
24      * @param max
25      * @param function
26      */
27     public Bisection(double a, double b, double tolerance, int max, F function) {
28         this.a = a;
29         this.b = b;
30         this.p = a + (b - a) / 2.0;
31         this.tolerance = tolerance;
32         this.max = max;
33         this.f = function;
34         this.currentI = 0;
35         this.finished = false;
36         this.DESRIPTION =
37             "This method will find a solution to  $f(x) = 0$ "
38             + " given the continuous function  $f$  on the"
39             + " interval  $[a, b]$  where  $f(a)$  and  $f(b)$  have opposite signs.";
40     }
41
42     @Override
43     public void step() {
44         if (currentI < max && finished != true) {
45             if (Math.abs(b - a) < (2 * tolerance)) {
46                 finished = true;

```

```

47         foundSolution = true;
48         return;
49     }
50     if (f.eval(p) <= 0)
51         a = p;
52     else
53         b = p;
54
55     p = a + (b - a) / 2;
56
57     ++currentI;
58 } else {
59     finished = true;
60 }
61 }
62
63 @Override
64 public String getDescription() {
65     return DESCRIPTION;
66 }
67
68 @Override
69 public void back() {
70     // TODO Auto-generated method stub
71 }
72
73 @SuppressWarnings("boxing")
74 @Override
75 public String printStep() {
76     return String.format("\n%d_%12f_%12f_%12f_%12f", currentI, a, b, p, f.eval(p));
77 }
78
79 /**
80  * @param args
81  */
82 @SuppressWarnings("boxing")
83 public static void main(String args[]) {
84     double a = 0, b = 0, tol = 0;
85     int max = 0;
86     F f = F.Poly;
87     if (args.length == 5) {
88         a = Double.parseDouble(args[0]);
89         b = Double.parseDouble(args[1]);
90         tol = Double.parseDouble(args[2]);
91         max = Integer.parseInt(args[3]);
92         f = F.valueOf(args[4]);
93     } else if (args.length == 4) {
94         a = Double.parseDouble(args[0]);
95         b = Double.parseDouble(args[1]);

```

```

96         tol = Double.parseDouble(args[2]);
97         max = Integer.parseInt(args[3]);
98     } else {
99         printUsageAndExit();
100    }
101    Bisection bisection = new Bisection(a, b, tol, max, f);
102    System.out.println(bisection.getDescription());
103    System.out.printf("%s:_%10s:_%10s:_%10s:_%10s:\n", "i", "a", "b", "p", "f(p)");
104    System.out.printf("_____");
105
106    Long startTime = System.currentTimeMillis();
107    while (!bisection.finished()) {
108        System.out.print(bisection.printStep());
109        bisection.step();
110    }
111    Long endTime = System.currentTimeMillis();
112    if (bisection.foundSolution())
113        System.out.println("\n\nDone\n0_found_at:" + bisection.getP());
114    else
115        System.out.println("\n\nUnable to find result in" + bisection.getMax()
116            + " iterations.");
117
118    System.out.println("Execution took:" + (endTime - startTime) + " ms and"
119        + bisection.currentI + " iterations");
120 }
121
122 public static void printUsageAndExit() {
123     System.out.println("Usage: _java _Bisection _left _right _tolerance _iterations <function>");
124     System.out.println("Optional _parameter _functions _can _be:");
125     for (F f : F.values())
126         System.out.println(f.name() + " _==_" + f.getDescription());
127     System.exit(0);
128 }
129
130 /**
131  * @return the a
132  */
133 public double getA() {
134     return a;
135 }
136
137 /**
138  * @param a
139  *         the a to set
140  */
141 public void setA(double a) {
142     this.a = a;
143 }
144

```

```

145  /**
146   * @return the b
147   */
148  public double getB() {
149      return b;
150  }
151
152  /**
153   * @param b
154   *         the b to set
155   */
156  public void setB(double b) {
157      this.b = b;
158  }
159
160  /**
161   * @return the tolerance
162   */
163  public double getTolerance() {
164      return tolerance;
165  }
166
167  /**
168   * @param tolerance
169   *         the tolerance to set
170   */
171  public void setTolerance(double tolerance) {
172      this.tolerance = tolerance;
173  }
174
175  /**
176   * @return the p
177   */
178  public double getP() {
179      return p;
180  }
181
182  /**
183   * @param p
184   *         the p to set
185   */
186  public void setP(double p) {
187      this.p = p;
188  }
189
190  /**
191   * @return the max
192   */
193  public int getMax() {

```

```
194         return max;
195     }
196
197     /**
198     * @param max
199     *         the max to set
200     */
201     public void setMax(int max) {
202         this.max = max;
203     }
204 }
```

Newtonson.java

```
1 public class Newtonson extends AbstractMethod {
2     private double start      = 0.5;
3     private double tolerance  = 1e-14;
4     private int    maxIterations = DEFAULT_MAX_ITERATIONS;
5     public F       f           = F.Poly;
6
7     public Newtonson(double start, double tolerance, int maxIterations, F f) {
8         this.start = start;
9         this.tolerance = tolerance;
10        this.maxIterations = maxIterations;
11        this.currentI = 1;
12        this.f = f;
13    }
14
15    @Override
16    public void back() {
17        // TO-DO
18    }
19
20    @Override
21    public String getDescription() {
22        return null;
23    }
24
25    @Override
26    public String printStep() {
27        return String.format("\n%d_%.12f", currentI, start);
28    }
29
30    @Override
31    public void step() {
32        if (currentI < maxIterations && finished != true) {
33            if (Math.abs(f.eval(start)) < tolerance) {
34                foundSolution = true;
35                finished = true;
36                return;
37            }
38            start = start - f.eval(start) / f.deriv(start);
39            currentI++;
40        } else
41            finished = true;
42    }
43
44    public static void main(String args[]) {
45        if (args.length != 4) {
46            printUsageAndExit();
47        }
48        Newtonson newton =
```



```

49         new Newtons(Double.parseDouble(args[0]), Double.parseDouble(args[1]), Int
50             .parseInt(args[2]), F.valueOf(args[3]));
51     Long startTime = System.currentTimeMillis();
52     while (!newton.finished()) {
53         System.out.print(newton.printStep());
54         newton.step();
55     }
56     Long endTime = System.currentTimeMillis();
57
58     if (newton.foundSolution())
59         System.out.println("\n\nDone.\n\nfound at:" + newton.getStart());
60     else
61         System.out.println("\n\nUnable to find result in" + newton.getMaxIterations());
62
63     System.out.println("Execution took:" + (endTime - startTime) + " ms and"
64         + newton.getCurrentI() + " iterations");
65
66 }
67
68 public static void printUsageAndExit() {
69     System.out.println(" Usage: _java _Newton _start _tolerance _iterations _<function>");
70     System.out.println(" Optional _parameter _functions _can _be:");
71     for (F f : F.values())
72         System.out.println(f.name() + " _==_" + f.getDescription());
73     System.exit(0);
74 }
75
76 /**
77  * @return the start
78  */
79 public double getStart() {
80     return start;
81 }
82
83 /**
84  * @param start
85  *         the start to set
86  */
87 public void setStart(double start) {
88     this.start = start;
89 }
90
91 /**
92  * @return the tolerance
93  */
94 public double getTolerance() {
95     return tolerance;
96 }
97

```

```

98      /**
99      * @param tolerance
100     *         the tolerance to set
101     */
102     public void setTolerance(double tolerance) {
103         this.tolerance = tolerance;
104     }
105
106     /**
107     * @return the maxIterations
108     */
109     public int getMaxIterations() {
110         return maxIterations;
111     }
112
113     /**
114     * @param maxIterations
115     *         the maxIterations to set
116     */
117     public void setMaxIterations(int maxIterations) {
118         this.maxIterations = maxIterations;
119     }
120 }

```

# Secant.java

```

1  /**
2   * @author Randall Hunt
3   */
4  public class Secant extends AbstractMethod {
5      private double tolerance;
6      private double x0;
7      private double x1;
8      private double x;
9      private double q0;
10     private double q1;
11     private int     maxIterations;
12
13     public F         f;
14
15     public Secant(double x0, double x1, double tolerance, int maxIterations, F f) {
16         this.x0 = x0;
17         this.x1 = x1;
18         this.tolerance = tolerance;
19         this.maxIterations = maxIterations;
20         this.currentI = 2;
21         this.f = f;
22         this.q0 = f.eval(x0);
23         this.q1 = f.eval(x1);
24     }
25
26     /**
27      * @param args
28      */
29     public static void main(String[] args) {
30         if (args.length != 5) {
31             printUsageAndExit();
32         }
33         Secant secant =
34             new Secant(Double.parseDouble(args[0]), Double.parseDouble(args[1]), Dou
35                 .parseDouble(args[2]), Integer.parseInt(args[3]), F.valueOf(args
36 Long startTime = System.currentTimeMillis();
37         while (!secant.finished()) {
38             System.out.println(secant.printStep());
39             secant.step();
40         }
41         Long endTime = System.currentTimeMillis();
42
43         if (secant.foundSolution())
44             System.out.println("\n\nDone. \n\nFound at: " + secant.getX1());
45         else
46             System.out.println("\n\nUnable to find result in " + secant.getMaxIterations
47
48         System.out.println("Execution took: " + (endTime - startTime) + " ms and")

```

```

49         + secant.getCurrentI() + "_iterations");
50     }
51
52     public static void printUsageAndExit() {
53         System.out.println(" Usage: _java _Secant _x0 _x1 _tolerance _iterations _<function>");
54         System.out.println(" Optional _parameter _functions _can _be:");
55         for (F f : F.values())
56             System.out.println(f.name() + " _==_" + f.getDescription());
57         System.exit(0);
58     }
59
60     @Override
61     public void back() {
62         // TODO
63     }
64
65     @Override
66     public String getDescription() {
67         // TODO Auto-generated method stub
68         return null;
69     }
70
71     @Override
72     public String printStep() {
73         return String.format("\n%d_%.15f_%.15f", currentI, x1, f.eval(x1));
74     }
75
76     @Override
77     public void step() {
78         x = x1 - q1 * (x1 - x0) / (q1 - q0);
79         if (currentI < maxIterations && finished != true) {
80             if (Math.abs(x - x1) < tolerance) {
81                 foundSolution = true;
82                 finished = true;
83                 return;
84             }
85             currentI++;
86             x0 = x1;
87             q0 = q1;
88             x1 = x;
89             q1 = f.eval(x);
90         } else
91             finished = true;
92     }
93
94     /**
95      * @return the tolerance
96      */
97     public double getTolerance() {

```

```

98         return tolerance;
99     }
100
101     /**
102     * @param tolerance
103     *         the tolerance to set
104     */
105     public void setTolerance(double tolerance) {
106         this.tolerance = tolerance;
107     }
108
109     /**
110     * @return the x0
111     */
112     public double getX0() {
113         return x0;
114     }
115
116     /**
117     * @param x0
118     *         the x0 to set
119     */
120     public void setX0(double x0) {
121         this.x0 = x0;
122     }
123
124     /**
125     * @return the x1
126     */
127     public double getX1() {
128         return x1;
129     }
130
131     /**
132     * @param x1
133     *         the x1 to set
134     */
135     public void setX1(double x1) {
136         this.x1 = x1;
137     }
138
139     /**
140     * @return the maxIterations
141     */
142     public int getMaxIterations() {
143         return maxIterations;
144     }
145
146     /**

```

```

147      * @param maxIterations
148      *           the maxIterations to set
149      */
150  public void setMaxIterations(int maxIterations) {
151      this.maxIterations = maxIterations;
152  }
153 }

```

## REFERENCES

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