Isaac Newtons Iterations-verfahren

Dokumentation

Von Francesco Ruiu &
Juan Antonio Agudo

Gruppe F Grün
WS 05/06

1. Entwicklung

Das Newtonsche Näherungsverfahren dient in erster Linie zur Bestimmung von Nullstellen von Funktionen. Es wurde von Isaac Newton am Ende des 17. Jarhhundert entwickelt. Zu dieser Zeit waren die Mathematischen Notationsverfahren zwar bereits stark formalisiert und verallgemeinert, jedoch wurden Beweise und dergleichen häufig ausschließlich geometrisch geführt.

Dies wird für heutige Mathematiker schnell recht komplex, da es nicht mehr üblich ist umfangreichere Systeme auf diese Art zu beweisen. Richard Feynmann, so heißt es, habe sich während eines Sabbaticals die Beweise zur Newtonschen Gravitationstheorie angesehen und dieser nicht nachvollziehen können.

Glücklicherweise Ist das Iterationsverfahren geometrisch, wie numerisch recht einfach zu begreifen. Es wurde erstmals in Newtons Schrift „Methodus fluxionum et serierum infinitarum“ veröffentlicht, welche uns leider nicht vorliegt, später fasste Newton sein Werk in der berühmten „Principia Mathematica“ zusammen, welche u.a. auch zum ersten mal die Integralrechnung beschreibt.

2. Prinzip des Näherungsverfahrens

Das Näherungsverfahren macht sich die Konvergenz von Folgen zunutze um ein Polynomiale Gleichung zu lösen. Man verwendet einen möglichst guten Erwartungswert für die Lösung des Polynoms und kommt durch die Iteration immer näher in die nähe der tatsächlichen Lösung, der Nullstelle der Funktion.

Die rekursive Definition lautet

$$x_{n+1} = Nf(x_n) = x_n - \frac{f(x_n)}{f'(x_n)}$$

Wobei x_0 der geschätzte Anfangswert ist, in dessen Nähe man die Lösung vermutet. Dieses Verfahren funktioniert je nach verwendeter Funktion mit quadratischer oder linearer Konvergenz.

3. Anwendungen

Obwohl Newton dieses Verfahren ursprünglich nur dafür einsetzte um Polynome zu lösen so wird es auf für folgende Anwendungen gebraucht:

1. Berechnung der Quadratwurzel (Verfahren nach Heron von Alexandria)
2. Berechnung des Schnittpunktes zweier Funktionen

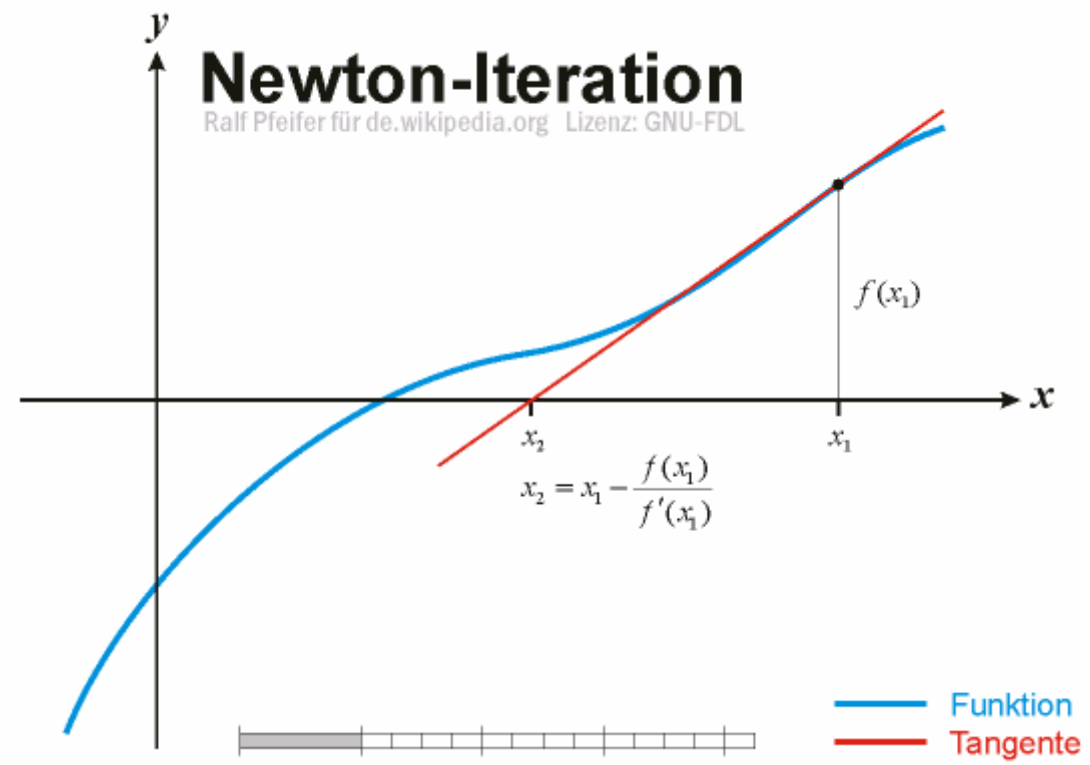
Im folgenden wird ein Java code gezeigt, der mit Hilfe der Newtonschen Iteration die Quadratwurzel zieht.

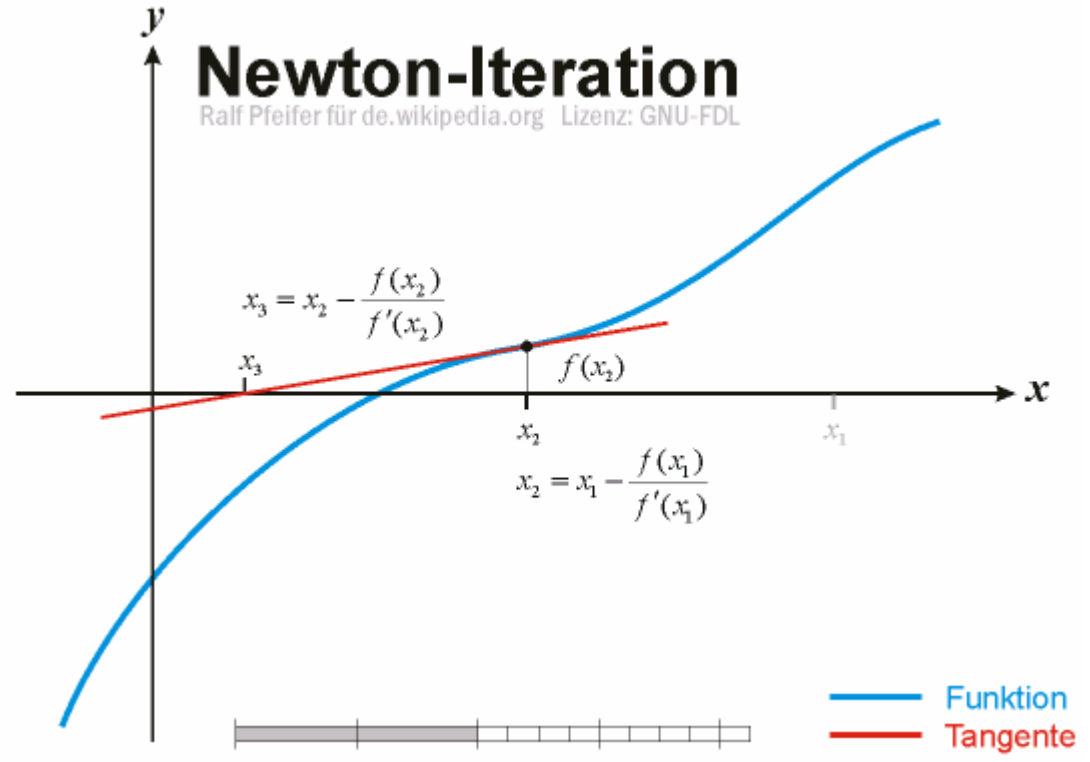
```
        public static BigDecimal newtSqrt(BigDecimal x, int scale){
                String xstr = x.toString();
                double x0 = Math.sqrt( Double.parseDouble( x.toString().substring(
                                                0,xstr.length() > 10 ? 10 : xstr.length())));
                BigDecimal xn = new BigDecimal(x0+"");
                xn.setScale(scale);
                BigDecimal lastVal=null;
                for (int i = 0; i < scale; i++) {
                        if(lastVal == null){
                                lastVal = xn;
                        }
                        xn = xn.subtract( xn.pow(2).subtract( x).
                                        divide(xn.multiply(TWO),scale,BigDecimal.ROUND_DOWN));
                        if(lastVal.equals(xn)){
                                return xn;
                        }else{
                                lastVal = xn;
                        }
                }
                return xn;
        }
```

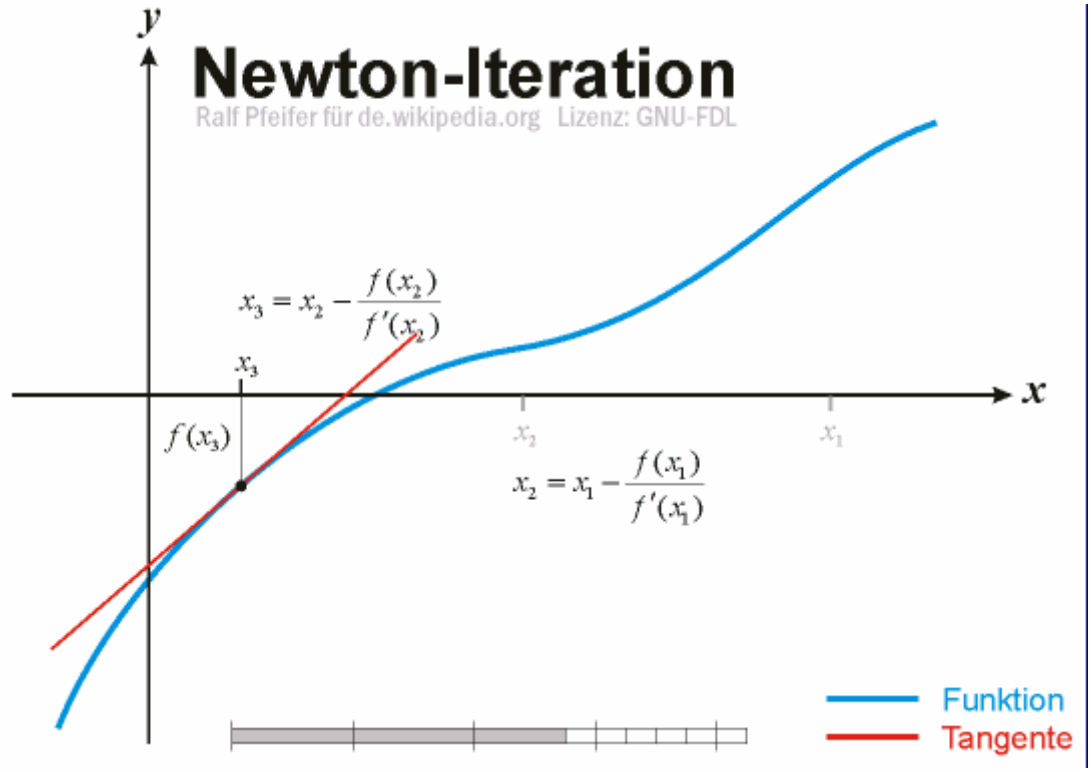
Dies ist hilfreich, wenn man z.B. eine beliebig genaue Annäherung an π berechnen möchte:

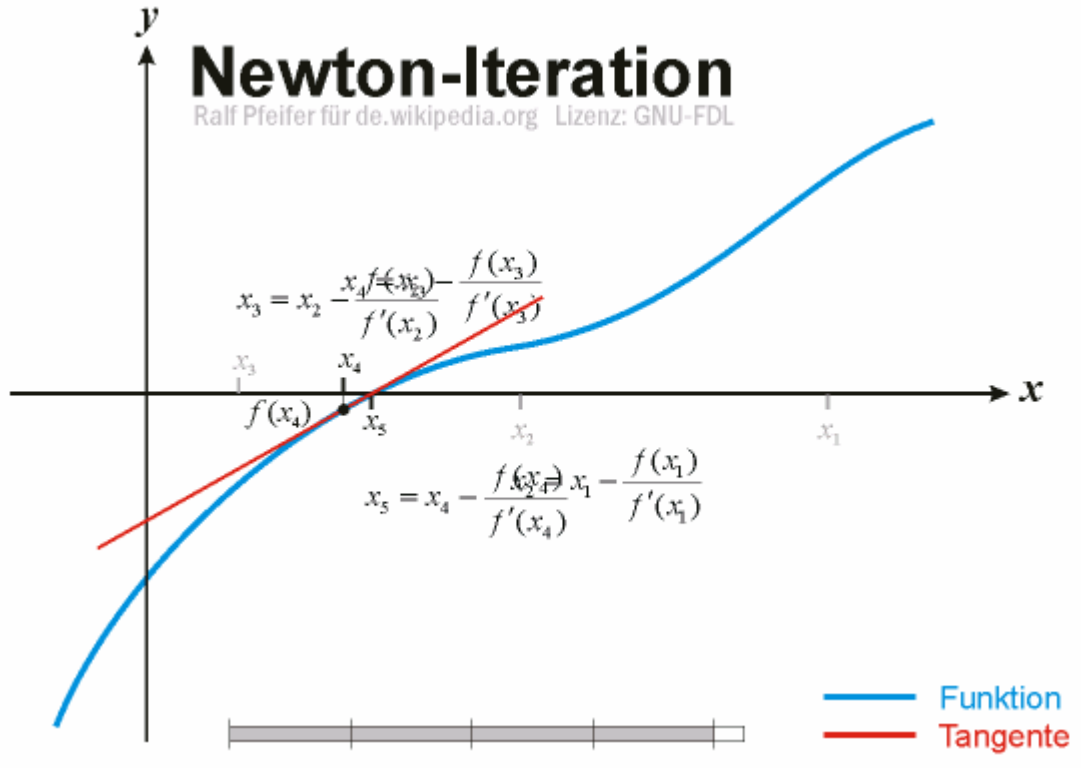
```
        public static final BigDecimal ZERO = new BigDecimal("0");
        public static final BigDecimal ONE  = new BigDecimal("1");
        public static final BigDecimal TWO  = new BigDecimal("2");

        public static void main(String[] args) {
                int precision = 512;
                int scale = precision;
                long start = System.currentTimeMillis();
                long end =0;
                BigDecimal a = ONE;
                BigDecimal b = ONE.divide(newtSqrt(TWO,scale),scale,BigDecimal.ROUND_DOWN);
                BigDecimal t = new BigDecimal("0.25");
                BigDecimal p = ONE;

                for (int i = 0; i < precision; i++) {
                        BigDecimal x = a.add(b).divide(TWO);
                        BigDecimal y = newtSqrt(a.multiply(b),scale/2);
                        BigDecimal ax = a.subtract(x);
                        ax = ax.pow(2);
                        t = t.subtract(p.multiply(ax));
                        a = x;
                        b = y;
                        p = TWO.multiply(p);
                }
                end = System.currentTimeMillis();
                System.out.println(end-start+"ms");
                BigDecimal pi = a.add(b).pow(2).divide(t.multiply(new
BigDecimal(4)),precision,BigDecimal.ROUND_UP);
                System.out.println(pi);
        }
```

4. Graphische Interpretation



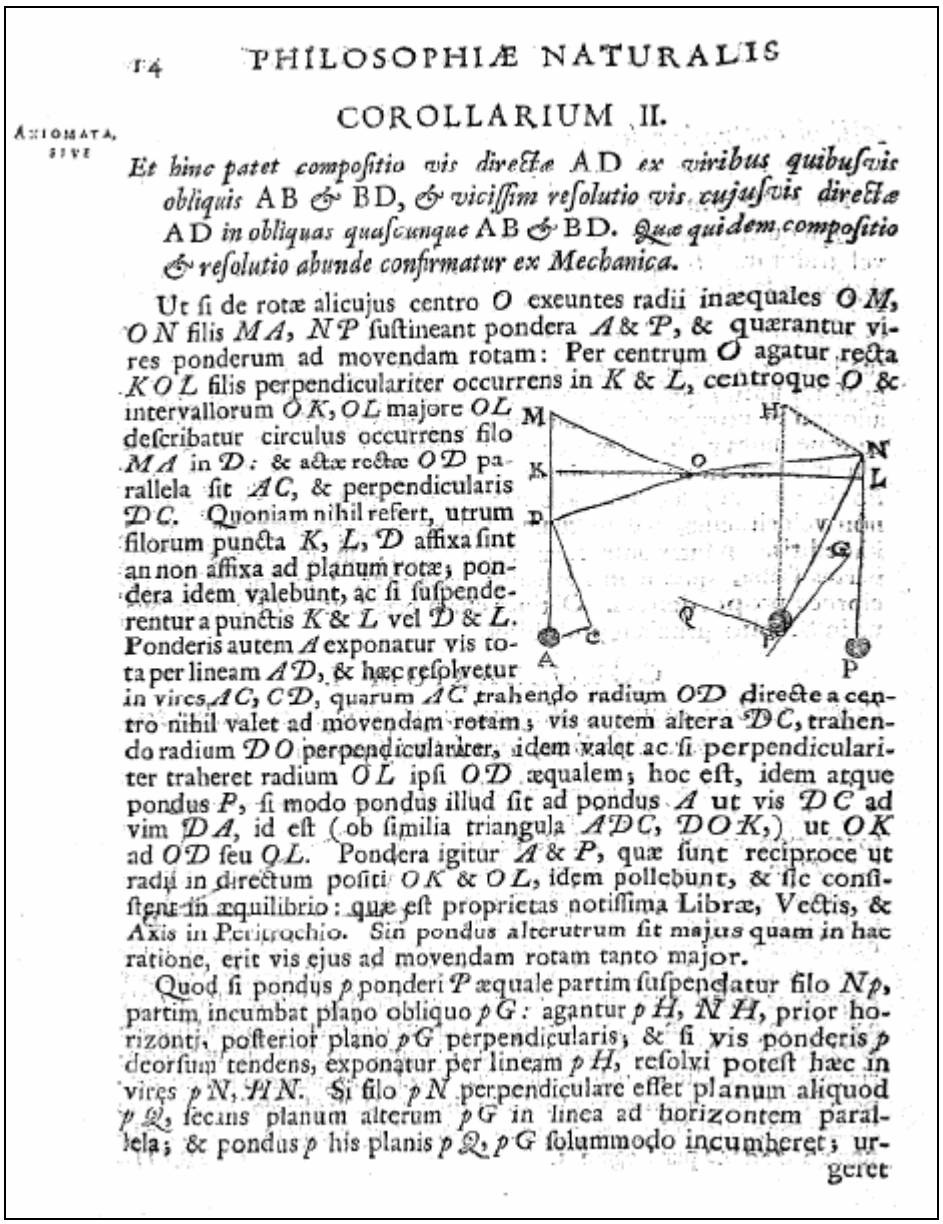



Newton-Iteration
Ralf Pfeifer für de.wikipedia.org Lizenz: GNU-FDL
y
x
$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$
x_3
$f(x_3)$
x_2
x_1
$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$
Funktion
Tangente


Newton-Iteration
Ralf Pfeifer für de.wikipedia.org Lizenz: GNU-FDL
y
x
$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$
x_4
x_3
$f(x_4)$
x_5
x_2
x_1
$x_5 = x_4 - \frac{f(x_4)}{f'(x_4)}$
$x_1 - \frac{f(x_1)}{f'(x_1)}$
Funktion
Tangente

5. Literaturdiskussion

Das Iterationsverfahren, welches wie oben bereits erwähnt sowohl in „Methodus fluxionum et serierum infinitarum“ als auch in der „Principia Matematica“ veröffentlicht wurde. Leider ist die Originalschrift in Latein verfasst und wir waren nicht in der Lage ihn in dem über 1000 Seiten starken Werk aufzufinden. Einen Auszug der „Principia Matematica“findet sich auf folgender Abbildung:



14 PHILOSOPHIÆ NATURALIS

AXIOMATA, SIVE

COROLLARIUM II.

Et hinc patet compoſitio vis directæ AD *ex viribus quibuſvis obliquis* AB *&* BD, *& viciſſim reſolutio vis cujuſvis directæ* AD *in obliquas quaſcunque* AB *&* BD. *Quæ quidem compoſitio & reſolutio abunde confirmatur ex Mechanica.*

Ut ſi de rotæ alicujus centro O exeuntes radii inæquales OM, ON filis MA, NP ſuſtineant pondera A & P, & quærantur vires ponderum ad movendam rotam: Per centrum O agatur recta KOL filis perpendiculariter occurrens in K & L, centroque O & intervallorum OK, OL majore OL deſcribatur circulus occurrens filo MA in D: & actæ rectæ OD parallela ſit AC, & perpendicularis DC. Quoniam nihil refert, utrum filorum puncta K, L, D affixa ſint an non affixa ad planum rotæ, pondera idem valebunt, ac ſi ſuſpenderentur a punctis K & L vel D & L. Ponderis autem A exponatur vis tota per lineam AD, & hæc reſolvetur in vires AC, CD, quarum AC trahendo radium OD directe a centro nihil valet ad movendam rotam; vis autem altera DC, trahendo radium DO perpendiculariter, idem valet ac ſi perpendiculariter traheret radium OL ipſi OD æqualem; hoc eſt, idem atque pondus P, ſi modo pondus illud ſit ad pondus A ut vis DC ad vim DA, id eſt (ob ſimilia triangula ADC, DOK,) ut OK ad OD ſeu OL. Pondera igitur A & P, quæ ſunt reciproce ut radii in directum poſiti OK & OL, idem pollebunt, & ſic conſiſtent in æquilibrio: quæ eſt proprietas notiſſima Libræ, Vectis, & Axis in Peritrochio. Sin pondus alterutrum ſit majus quam in hac ratione, erit vis ejus ad movendam rotam tanto major.

Quod ſi pondus p ponderi P æquale partim ſuſpendatur filo Np, partim incumbat plano obliquo pG: agantur pH, NH, prior horizonti, poſterior plano pG perpendicularis; & ſi vis ponderis p deorſum tendens, exponatur per lineam pH, reſolvi poteſt hæc in vires pN, HN. Si filo pN perpendiculare eſſet planum aliquod pQ, ſecans planum alterum pG in linea ad horizontem parallela; & pondus p his planis pQ, pG ſolummodo incumberet; urgeret

Die Funktionsweise des Algorithmus suchten wir also aus frei verfügbaren Quellen im Internet zusammen. Hierzu:

http://de.wikipedia.org/wiki/Newton_Iteration
http://en.wikipedia.org/wiki/Newton%27s_method
http://en.wikipedia.org/wiki/Methods_of_computing_square_roots

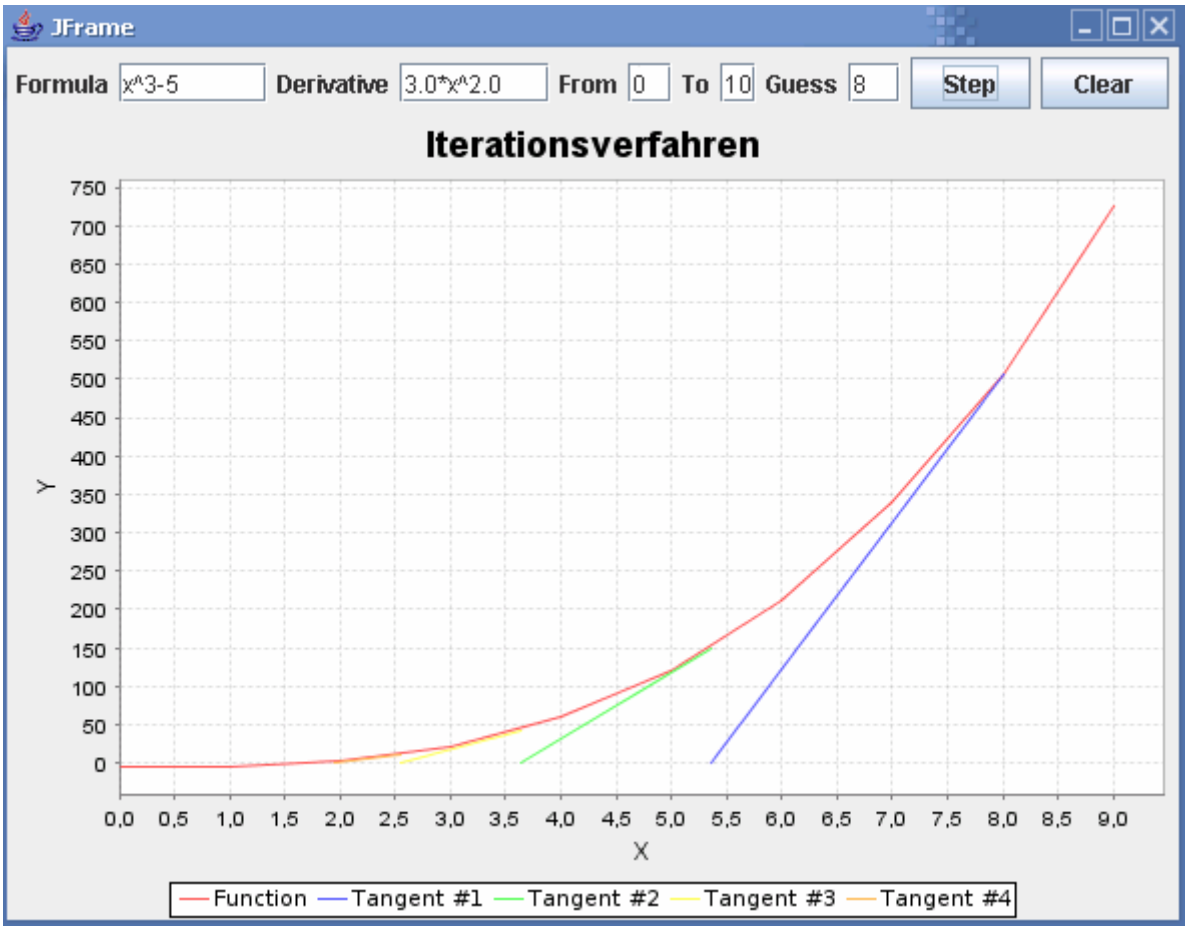
6. Implementierung einer graphischen Lösung des Verfahrens

Um eine anschauliche Lösung des Verfahrens präsentieren zu können entschlossen wir uns eine Swing GUI zu schreiben in der die angebrachten Tangenten interaktiv angebracht und betrachtet werden können.

Funktionsweise:

Zunächst trägt man eine Formel in das entsprechende Feld ein, das eintragen der ersten Ableitung ist nicht notwendig, da sie beim ersten Schritt automatisch berechnet wird. Der Wertebereich in dem der entstehende Graph gezeigt werden soll muss nun noch genau so werden, wie die Stelle an der man in etwa die Nullstelle vermutet.

Wir verwendeten das JEP Framework (http://www.singularsys.com/jep/) um die eingefügten Funktionen zu evaluieren und zu differenzieren.



Im Folgenden wird der Code aufgeführt, der Druck auf den „Step“ Button ausgeführt wird. Er enthält die benutzten Mathematischen Grundlagen die das Iterationsverfahren ausmachen. Als auch eine Utility Klasse, die sich um die Ableitung der Funktion kümmert.

```
public void actionPerformed(java.awt.event.ActionEvent e) {
	String formula = textFormula.getText();
	 String derivative = null;

	try {
		jep.addStandardConstants();
		jep.addStandardFunctions();
		jep.addComplex();
		jep.setAllowUndeclared(true);
		jep.setAllowAssignment(true);
		jep.setImplicitMul(true);

		// Sets up standard rules for differentiating sin(x) etc.
		jep.addStandardDiffRules();
		Node node = jep.parse(formula);

		Node diff = jep.differentiate(node,"x");
		diff = jep.simplify(diff);
		derivative = jep.toString(diff);

		textDerivative.setText(derivative);
	} catch (ParseException e1) {
		// TODO Auto-generated catch block
		e1.printStackTrace();
	}

	if(step==0){
		lastX = Double.parseDouble(textGuess.getText());
		XYSeries functionSeries = new XYSeries("Function");
		Integer fromVal = Integer.parseInt(textFrom.getText());
		Integer toVal = Integer.parseInt(textTo.getText());
		jep.parseExpression(formula);

		for (int i = fromVal; i < toVal; i++) {
			jep.addVariable("x",i);
			functionSeries.add(i,jep.getValue());
		}
		collection.addSeries(functionSeries);
	}else{
		XYSeries newLine = new XYSeries("Tangent #"+step);
		jep.parseExpression(formula);
		jep.addVariable("x",lastX);

		double y1 = jep.getValue();
		newLine.add(lastX,y1);


		jep.parseExpression(formula);
		jep.addVariable("x",lastX);
		double fx1 = jep.getValue();

		jep.parseExpression(derivative);
		jep.addVariable("x",lastX);
		double fderiv1 = jep.getValue();

		double tmp = fx1/fderiv1;
		lastX = lastX-tmp;
		System.out.println(lastX);
		newLine.add(lastX,0);
		collection.addSeries(newLine);
	}
	step++;
}
```

7. Ergebnisse

Die Implementierung des Iterationsverfahrens in einer Programmiersprache ist recht einfach und die Umstellung der Formel erlaubt eine recht flexible Verwendung.

Die Konvergenzkriterien mit denen der Algorithmus gegen den Nullpunkt strebt sind nicht fix, sondern unterscheiden sich in ihrer Qualität bei den verschiedenen Funktionen immens.