**Hochschule München**

Studiengang Informatik

**Angewandte Mathematik**

Übungsblatt 1

1. Grafische Darstellung von Funktionen
2. Nullstellen von nicht-linearen Gleichungen

Bastian [Kersting](https://moodle.hm.edu/user/view.php?id=72004&course=11061)

Michael Schober

Elena Lilova

IF2B

**Aufgabe 1**

1. Die Aufgabe ist es zwei Funktionen zu zeichnen. Der Wertebereich wird auf das Intervall von null bis zehn beschränkt:

Zuerst definieren wir den Wertebereich von 1e-3 bis 10 auf der x-Achse:

*x = np.linspace(0.001,10,100)*

Danach werden noch die zwei Funktionen definiert:

*f1 = np.sqrt(x)*

*f2 = (np.log(x)) / (x+1)*

Jetzt können wir die Funktionen zeichnen. Wir wollen, dass alles leicht zu verstehen und deutlich lesbar wird, dafür brauchen wir einen Titel und die Achsenbezeichnungen. Das erreichen wir mit dem folgenden Code:

*ax.set\_title('Linear')*

*a.set\_xlabel("x-Axis")*

*a.set\_ylabel("y-Axis")*

Wenn wir uns auf die Formatierung gekümmert haben, es ist Zeit unsere zwei Funktionen zu zeichnen.

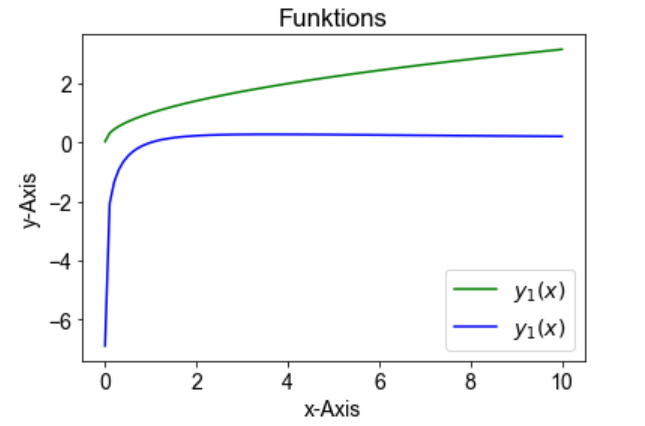
In Python wird das “plotten“ genannt. Die Methode, die wir nutzen um die Funktionen auf die Graph zu zeichnen ist: *plot().* Wir können beliebige Farbe wählen und mit den folgenden drei Zeilen Code kriegen wir unsere Funktionen gezeichnet:

*a.plot(x, f1, color="green",label="$y\_1(x)$")*

*a.plot(x, f2, color="blue",label="$y\_1(x)$")*

*plt.show()*

Das Ergebnis:



1. Die nächste Aufgabe ist ähnlich, jetzt zeichnen wir die folgende Funktion:

linear und halb-logarithmisch.

Dieses Mal wollen wir die Funktionen in zwei verschiedene Graphen zeigen. Wir definieren ob wir die beide Graphen neben- oder unter-einander darstellen wollen und können noch die Größe wählen:

*fig, axis1b = plt.subplots(1,2, figsize=(12, 3))*

In unseren Fall wir legen zwei Fenster fest, sie werden sich in eine Zeile befinden (nebeneinander).

So wie in der vorherigen Aufgabe definieren wir uns die Grenzwerte, benennen Sie und definieren die Funktion selbst:

*x = np.linspace(0,10,100)*

*f1 = np.exp(x)*

*axis1b[0].set\_title('Linear')*

*axis1b[1].set\_title('SemiLog')*

*axis1b[0]/axis1b[1]* – die Bezeichnungen in [] dienen uns zu definieren welches „Fenster“, also Achsenobjekt wir ansprechen.

Wir wollen noch die Achsen Beschreibungen fest legen. Das können wir entweder für jeden Graph einzeln machen, oder wir können uns das Leben mit einer Schleife erleichtern. In Python sieht das folgendermaßen so aus:

*for axis in axis1b:*

*axis.set\_xlabel("x-Axis")*

*axis.set\_ylabel("y-Axis")*

*axis.legend()*

Und wenn wir schon alles verschönert haben bleibt uns nur noch die Funktionen zu zeichnen. Für die Lineare Funktion benötigen wir wieder nur die plot() Methode und für die und halb-logarithmische Funktion können wir die Methode semilogy() benutzen. Wählen wir schöne Farben und „plotten“ wir alles raus mit der Methode show():

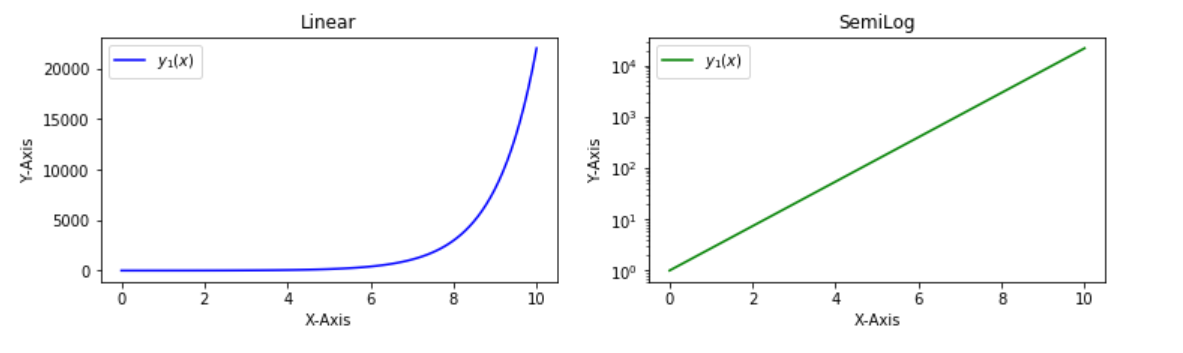
*axis[1].plot(x,f1, color = " blue", label="$y\_2(x)$")*

*axis[0].semilogy(x, f1, color = "green", label="$y\_1(x)$")*

*plt.show()*

Und voila so sehen unsere zwei Graphen aus:

Die Funktion ist halblogarithmisch als Gerade dargestellt. Die y Achse wir mit dem Befehl semilogy() formatiert.



1. In diese Aufgabe zeichnen wir dieses Mal eine Funktion linear und doppelt-logarithmisch:

Auch Teil der Aufgabe ist es zu zeigen, dass die Funktion:

In doppelt-logarithmischer Darstellung immer eine Gerade ergibt.

Ähnlich wie in Aufgabe 1b. haben wir mehr als einen Graphen. Dazu passen wir den Code aus Aufgabe 1b an. Wir ändern die Anzahl der „Fenster“ von zwei auf drei:

*fig, axis1c = plt.subplots(1, 3, figsize=(12, 3))*

Danach folgt wieder die Initialisierung und Formatierung, die wir schon bereits kennen:

*x = np.linspace(0, 10, 100)*

*n = np.linspace(-100,100,200)*

*axis1c[0].set\_title('LogLog')*

*axis1c[1].set\_title('Linear')*

*axis1c[2].set\_title('Proof')*

*for axis in axis1c:*

*axis.set\_xlabel("x-Axis")*

*axis.set\_ylabel("y-Axis")*

*axis.legend()*

Wir definieren auch die Funktion:

*f1 = np.sqrt(x)*

Jetzt können wir das erste Teil der Aufgabe machen und die gegebene Funktion einmal als doppelt-logarithmisch und einmal als Lineare Funktion zu zeigen. Für die doppelt-logarithmische Funktion verwenden wir die loglog() Methode:

*axis1c[0].loglog(x, f1, color="green", label="$y\_1(x)$")*

*axis1c[1].plot(x, f1, color="red", label="$y\_2(x)$")*

Jetzt kümmern wir uns um dem zweiten Teil der Aufgabe. Wir definieren die Funktion:

*f2 = x \*\* number*

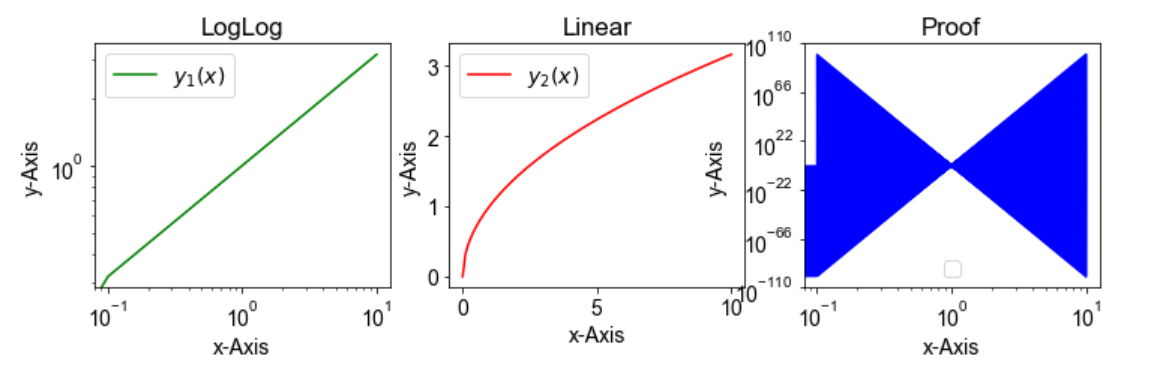
und machen eine Schleife, die durch aller Nummern geht und eine Gerade zeichnet. Wir haben „n“ im Wertebereich von -100 bis 100 definiert und zeigen Geraden für 200 verschiedene „numbers“ in diesem Wertebereich:

*for number in n:*

*f2 = x \*\* number*

*axis1c[2].loglog(x, f2, color="blue")*

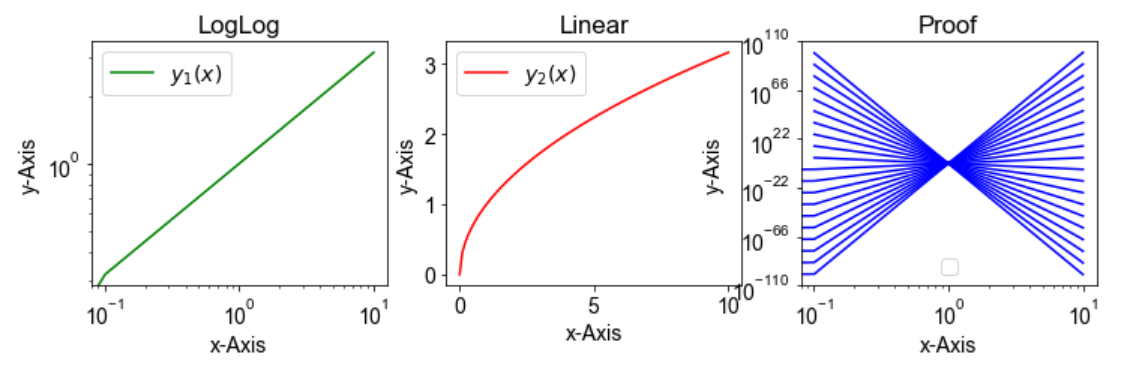
Jetzt plotten wir alle drei Graphen:



Wir könnten auch die Anzahl vom n reduzieren, damit wir die Geraden besser sehen können:

*n = np.linspace(-100,100,20)*

Die neue Grafik sieht jetzt so aus:



1. Hier ist die Gleichung einer zweiseitigen Hyperbel in Parameterdarstellung gegeben:

Wir wissen, dass *a* der Abstand der Brennpunkte vom Zentrum ist und *b* ist gegeben als:

(*e* ist die Exzentrizität)

Jetzt müssen die Äste der Hyperbel gezeichnet werden mit

a = 0.1 und

Jetzt fangen wir wieder mit den schon für uns bekannten Schritten an und definieren Grenzwerte, die gegebene Werte und Funktionen, setzten Titel, formatieren die Graphen. Unser Code sieht jetzt so aus:

*fig, ax = plt.subplots()*

*a = 0.1*

*e = 5*

*b = a \* (np.sqrt(e \*\* 2 - 1))*

*t = np.linspace(-2, 2, 100)*

*xPlus = np.cosh(t)*

*xMinus = - xPlus*

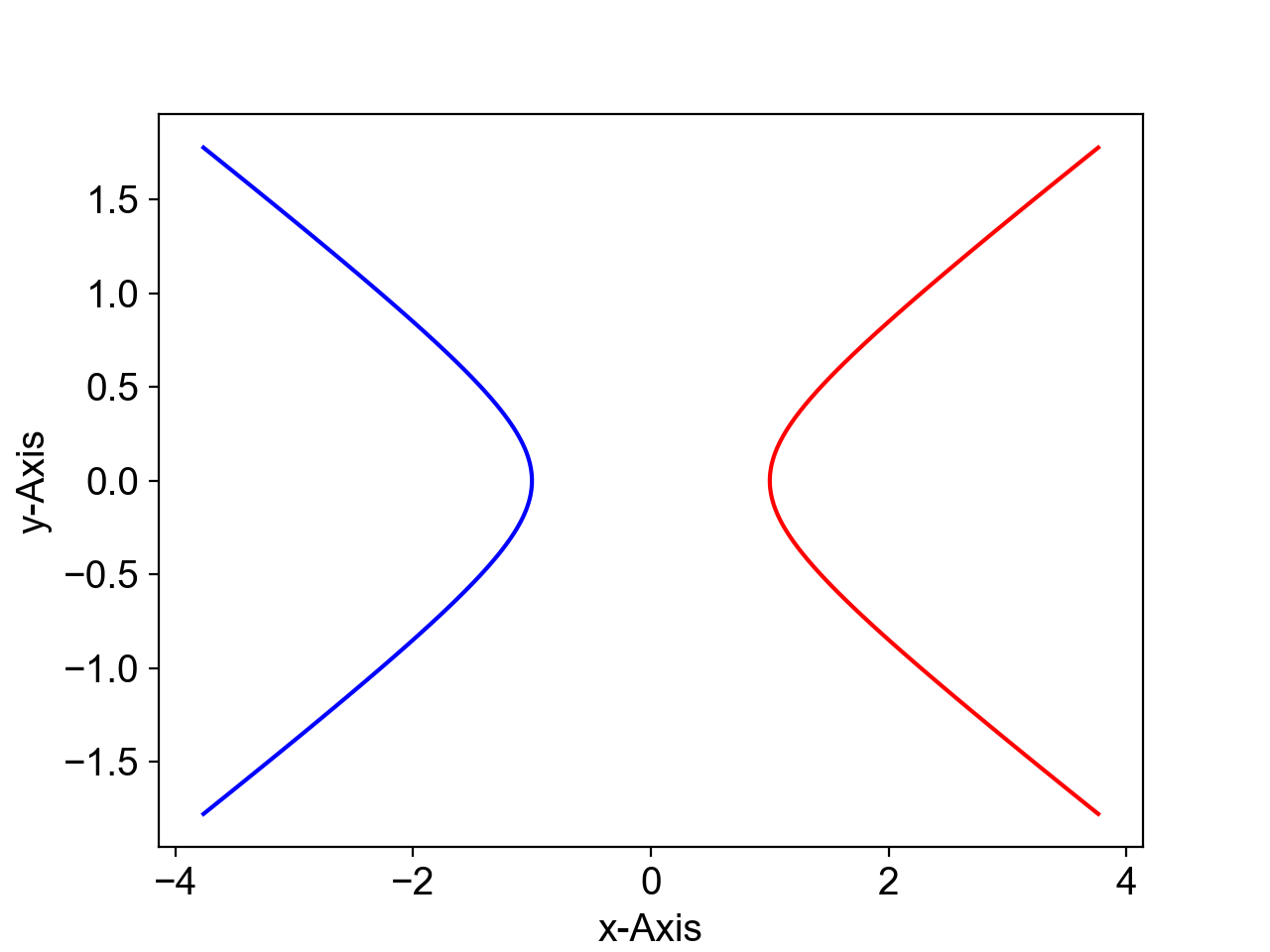
*y = b \* np.sinh(t)*

Jetzt fehlt es nur noch die Funktionen zu plotten. Wir definieren zwei Funktionen für den Kosinus hyperbolicus, weil wir Ihn einmal mit positiven Wert und einmal mit negativen Wert darstellen wollen.

*ax.plot(xPlus,y, color="red")*

*ax.plot(xMinus,y, color="blue")*

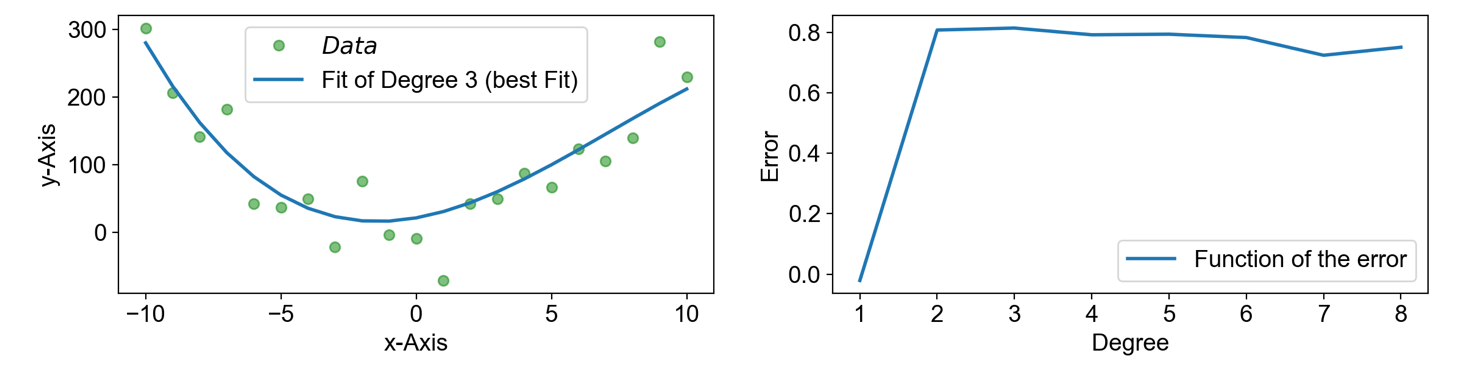
plt.show()



## e) Bei der linearen Regression geht es darum, die optimale Funktion durch eine gegebene Anzahl an Datenpunkten zu finden. Die Funktion sollte einen möglichst kleinen Fehler haben, aber auch nicht genau durch alle Punkte laufen (Interpolation), da sonst Prognosen deutlich ungenauer werden. Um die optimale Funktion zu finden benutzen wir das Verfahren der Kreuzvalidierung.

Wir haben zuerst unsere Funktion auf Basis jedes zweiten Datenpunktes, mithilfe der pylab.polyfit Funktion bestimmt. Nun wurde der Fehler, bezüglich aller Datenpunkte, mithilfe der folgenden Formel berechnet (R-squared Methode):

Dabei wird der Fehler zwischen Daten und Vorhersage berechnet und quadriert. Der Fehler bewegt sich damit zwischen einem Wert von null und eins, wobei eins als optimal und null als schlechtester Fit zu bewerten ist. Da wir unseren Fit nur auf Basis jedes zweiten Datenpunktes erstellt haben, können wir nun eine Bewertung hinsichtlich aller Datenpunkte treffen (Kreuzvalidierung) und eine Aussage über die Prognose der Funktion machen. Die Ergebnis haben wir hier aufbereitet:



Wir kamen zu folgendem Ergebnis: Die Funktion dritten Grades weißt den niedrigsten Fehler auf. Betrachtet man die Funktion des Fehlers, so fällt auf, dass R2 für eine Funktion ersten Grades (lineare Funktion) null ist und einen sehr schlechten Fit darstellt. Der beste Fit wird bei einer Funktion dritten Grades erreicht.

**Aufgabe 2**

(a) Lösen Sie folgende Gleichungen:

**i. cos(x) = x** <=> f(x) = x – cos(x)

***Sekantenverfahren:***

Für diesen Algorithmus müssen zunächst zwei Startpunkte a und b gewählt werden, wobei einer der zugehörigen Funktionswerte f(a) oder f(b) negativ sein muss, damit sich das Verfahren einer Nullstelle des Graphen nähern kann. Dabei wird die Sekante zwischen f(a) und f(b) gebildet. Der Schnittpunkt von Sekante und x-Achse markiert den Funktionswert f(x). Ist f(x) \* f(a) negativ, wird aus f(x) der neue Punkt f(b), durch den die nächste Sekante gebildet wird. Ist f(x) \* f(b) negativ, wird entsprechend f(a) angepasst. Besteht keine der zwei Bedingungen bricht das Verfahren ab. Durch dieses Verfahren befindet sich die anzunähernde Nullstelle stets zwischen den sich nähernden Funktionswerten f(a) und f(b). Die Berechnung des Schnittpunktes zwischen Sekante und x-Achse folgt der Gleichung:

x **=** a **-** f(a)**\***(b **-** a) **/** (f(b) **-** f(a))

Das Verfahren endet durch Abbruch oder durch erreichen der Nullstelle mit f(x) = 0 bzw. einer geeigneten Näherung.

***Ergebnis:*** 1 Nullstelle bei x = 0.7390851332151607

**ii. 2 sin(x) = x** <=> j(x) = x – 2 \* sin(x); dj(x) = 1 – 2 \* cos(x)

***Newton-Verfahren:***

Das Newton-Verfahren ähnelt dem Sekantenverfahren. Statt einer Sekante zwischen zwei Startpunkten wird jedoch die Tangente eines einzelnen Startpunktes a gebildet. Der Schnittpunkt x zwischen Tangente und x-Achse wird errechnet und dessen zugehöriger Funktionswert als neue Basis der Tangente eingesetzt. Die Iteration folgt der Formel:

a **=** x **–** f(x) **/** Df(x)

Das Verfahren bricht ab nachdem sich einer Nullstelle in vorgegebener Toleranz genähert wurde, oder nachdem die Anzahl der maximal auszuführenden Iterationen überschritten wurde. Die Toleranz stellt dabei die Höhe des aktuell iterierten Funktionswertes f(a) dar. Die obige Funktion besitzt drei Nullstellen, weshalb drei verschiedene Startwerte benötigt werden.

***Ergebnis****:* 3 Nullstellen bei x1 = -1.8954942672087132; x2 = 0; x3 = 1.8954945666276892

**iii. x = 2 – exp(-x)** <=> q(x) = (2 – exp(-x)) – x

***Bisektionsverfahren:***

Das Bisektionsverfahren ähnelt dem Sekantenverfahren und besitzt ebenfalls zwei Startwerte a und b. Statt der Sekante zwischen den zugehörigen Funktionswerten wird hier jedoch lediglich der Mittelwert auf der x-Achse zwischen a und b gebildet. Die Entscheidung ob a oder b nun durch den neuen Wert ersetzt wird erfolgt ebenfalls über die Bedingung f(a)\*f(x) < 0 oder f(b)\*f(x) < 0 (vgl. Sekantenverfahren). Die Iteration der Werte erfolgt über die Formel:

x = (a + b) / 2

Das Verfahren bricht ab sollte die Bedingung zur Anpassung der Grenzen nicht gegeben sein, oder durch Erreichen der Nullstelle, bzw. einer geeigneten Näherung. Um mehrere Nullstellen zu bestimmen, muss das Verfahren dementsprechend mit mehreren Startintervallen ausgeführt werden.

***Ergebnis:*** 2 Nullstellen bei x1 = -1.1461932206205825; x2 = 1.8414056604369606

**iv. x = exp(1 – x^2)** <=> p(x) = exp(1 – x^2) – x

Bei der letzten Gleichung wurde ebenfalls das bereits erklärte Bisektionsverfahren angewandt.

***Ergebnis:*** 1 Nullstelle bei x = 1

**(b)** In der Molekularfeldtheorie des Ferromagnetismus wird die Magnetisierung

m aus der Gleichung m = tanh(m/T) bestimmt. Bestimmen Sie die Lösung m (m != 0) als Funktion der Temperatur T mit einer Genauigkeit von 10^-6.

***Fixpunktiteration:***

Die Fixpunktiteration bestimmt schrittweise den Schnittpunkt der Funktion mit der Ursprungsgerade f(x) = x. In unserem Beispiel hat die Gleichung die Form m = tanh(m/T). Es wird ein Startwert für m gewählt. Dieser Startwert bildet einen neuen Funktionswert m der wiederum Iterativ in die Funktion eingesetzt wird und somit eine Folge von Werten für m bildet. Das Verfahren bricht ab, sobald zwei aufeinanderfolgende Werte für m eine Differenz von höchstens 10^-6 besitzen. Während eines Durchlaufs der Fixpunktiteration bleibt T konstant.

Um aber die Magnetisierung m als eine Funktion der Temperatur T abzubilden, muss die Fixpunktiteration mit verschiedenen Werten für T (zwischen 0 und 1) durchgeführt werden.

Hierfür wird für T der Startwert 0.01 gewählt. In einer Schleife wird T schrittweise um 0.01 erhöht und in einer Liste gespeichert. Dabei wird zu jedem T ein korrespondierender Wert m (mit entsprechender Toleranz) über Fixpunktiteration errechnet und ebenfalls in einer Liste gespeichert. Zuletzt werden die Werte der Temperatur auf die Werte der Magnetisierung geplottet und ergeben folgende ***Lösung:***

## d) Um die beiden Gleichungen zu lösen wird erst die eine Gleichung umgestellt und dann in die andere eingesetzt. Mithilfe der SymPy Bibliothek ist dies in einem Schritt mit der solve() Methode zu erreichen:

*x, y, a, b = symbols('x y a b')*

*solu = solve((b - a\*y - x\*\*2 \* y, -x + a \* y + x\*\*2 \* y ), (y, x))*

Nun kommen wir auf die beiden Lösungen:

ii) Die beiden Gleichungen werden umgestellt: Die zweite Gleichung lösen wir wieder mit SymPy nach auf:

*x,y,b,a = symbols("x y b a")*

*soly = solve(-x + a \* y + x\*\*2 \* y,y)*

Dies liefert folgendes Ergebnis:

Da wir wissen das ist kann wie folgt nach x aufgelöst werden:

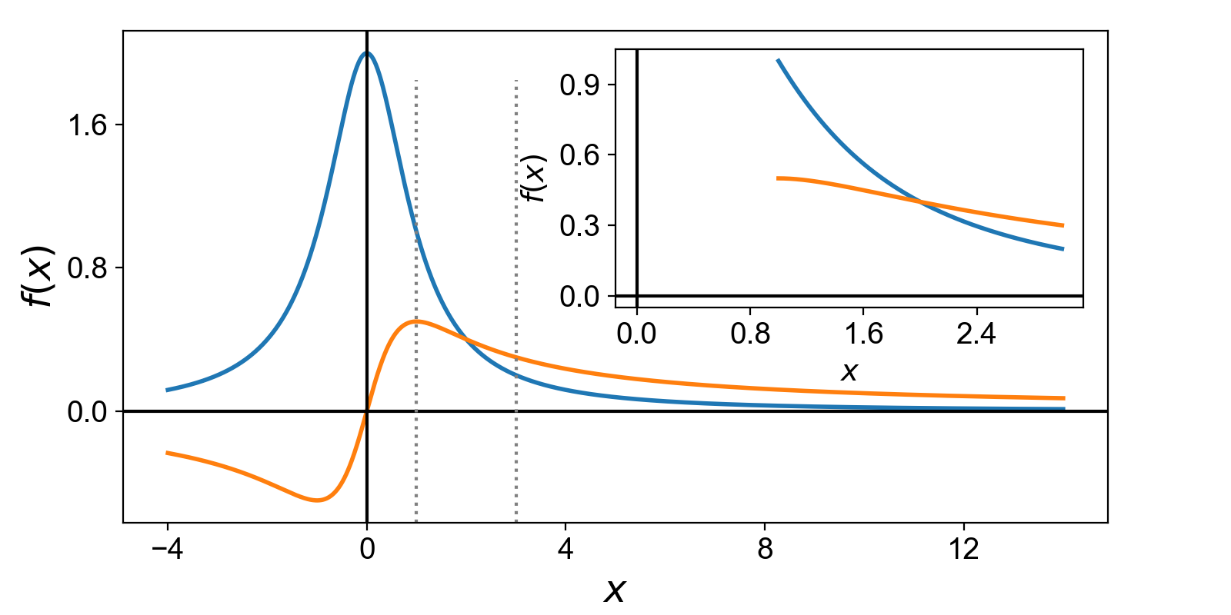
Bei einsetzen von y kürzt sich und es bleibt .

Die Fixpunktiteration muss nun neben einem x-Wert in jeder Iteration auch den y-Wert berechnen. Wenn wir die obigen Gleichungen in die Fixpunktiteration einsetzen, sehen wir dass das Ergebnis oszilliert, also immer zwischen null und neununddreißig hin- und herspringt. Die Gleichungen müssen also in eine andere Form gebracht werden:

Die erste Gleichung wird nach x aufgelöst, die zweite nach y. So ergeben sich folgende Ergebnisse:

Eingesetzt in die Fixpunktiteration ergeben sich folgende Lösungen:

Betrachten wir nun die Lösung grafisch, indem wir beide Gleichungen nach umstellen:



Die Lösung (2/0.4) ist plausibel.

## Um den Lagrange Punkt L1 zu finden wird die Gleichung mithilfe von SymPy nach aufgelöst:

*G,M,R,r,m,w = symbols('G,M,R,r,m,w')*

*sol = solve((G\*M)/(r\*\*2)-(G\*m)/((R-r)\*\*2),(w\*\*2 \* r), r)*

Dies liefert das Ergebnis:

Nun werden die gegebenen Variablen eingesetzt. Durch die Wurzel kommt es zu zwei verschiedenen Lösungen: