**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

**Einführung**

**Aufgabe 1**

1. Die Aufgabe ist es nun zwei Funktionen zu zeichnen. Die Grenzwerten der beide Funktionen liegen zwischen 0 und 10:



*fig, a = plt.subplots()*

Zuerst definieren wir die Grenzwerte 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 verstehe 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 vorherdefinierte Funktionen zu zeichnen.

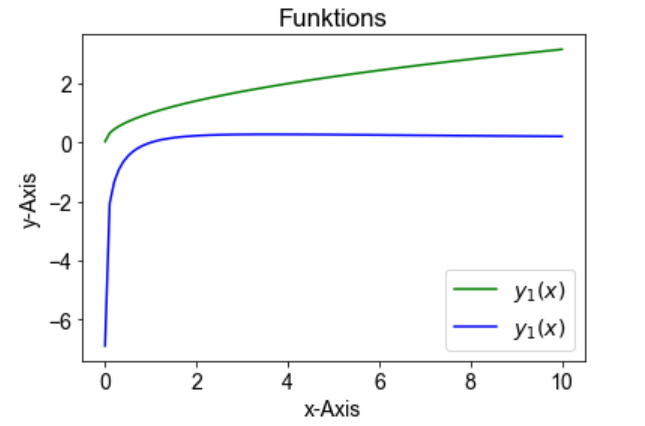
In Python wird das “ploten“ genannt. Die Methode, die wir nutzen um die Funktionen auf die Graph zu zeigen ist: *plot().* Wir können beliebige Farbe wählen und mit den folgenden drei Zeilen Code kriegen wir unsere Funktionen auf die Graph:

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

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

*plt.show()*

Und das End Ergebnis:



1. Die nächste Aufgabe ist es 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 naben- oder unter-einander wollen und können noch die Größe wählen:

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

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

So wie in der vorherigen Aufgabe definieren wir uns die Grenzwerte, schreiben Title 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 welche Graph sprechen wir an.

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")*

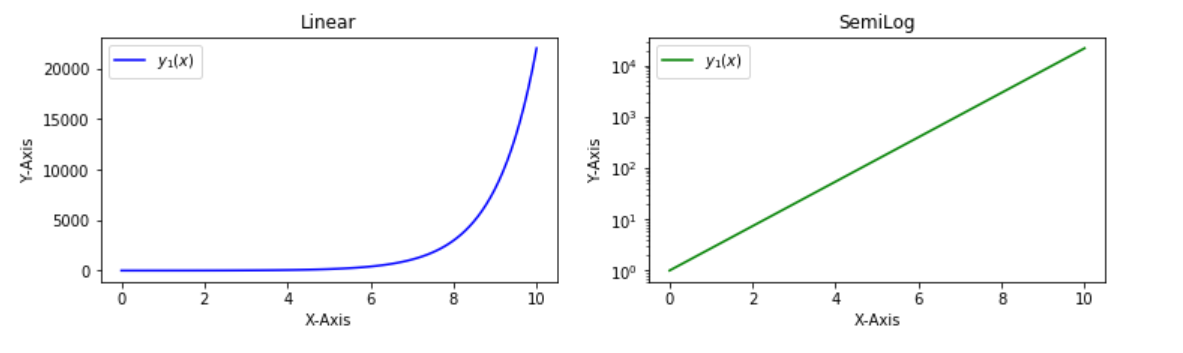
*axis.legend()*

Und wenn wir schon alles verschönert haben bleibt uns nur noch die Funktionen zu zeigen. 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 schow():

*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:

1. In diese Aufgabe zeigen wir dieses Mal eine Funktion Linear und doppelt-logarithmisch:



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



In doppelt-logarithmischer Darstellung immer eine Gerade ergibt.

Ähnlich wie in Aufgabe 1b. haben wir mehr als einem Graph. Jetzt brauchen wir nur wir unser Code von Aufgabe 1b ein bisschen anzupassen. Wir ändern die Graphen von 2 auf 3 und werden sie wieder in eine Zeile erzeugen:

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

Danach Folgt wieder die Initialisierung, 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. Definieren wir die Funktion:

*f2 = x \*\* number*

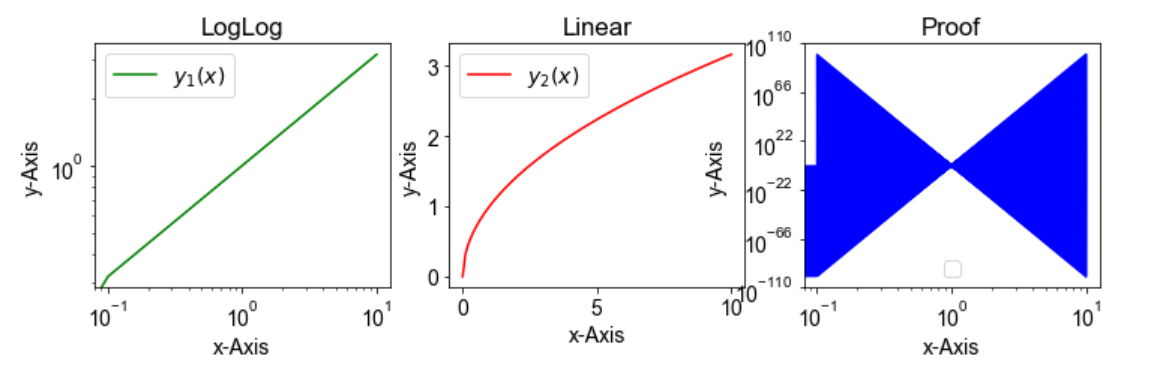
und machen eine Schleife, die durch aller Nummern geht und eine Gerade zeichnet. Wir haben „n“ im Grenzwert von -100 bis 100 definiert und zeigen Geraden für 200 verschiedene „number“ in diesem Grenzwert:

*for number in n:*

*f2 = x \*\* number*

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

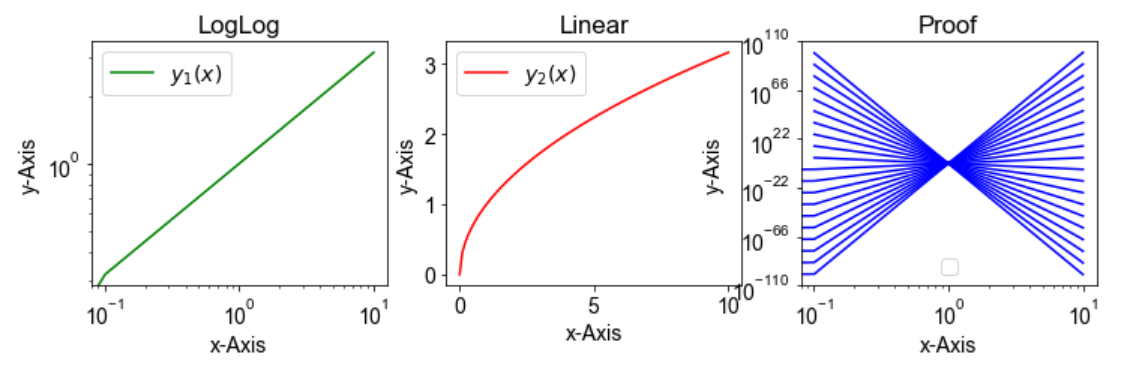
Jetzt zeigen wir alle drei Graphen:



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. In diesen Teil haben wir die Gleichung einer zweiseitigen Hyperbel in Parameterdarstellung gegeben:



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

 (*e* ist die Exzentrizität)

Jetzt müssen die Äste der Hyperbel gezeichnet werden mit

 und 

Jetzt fangen wir wieder mit den schon für uns bekannten Schritten an und definieren Grenzwerte, definieren die gegebene Werte und Funktionen, setzten Titel, Formatieren die Graphen und unserer 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*

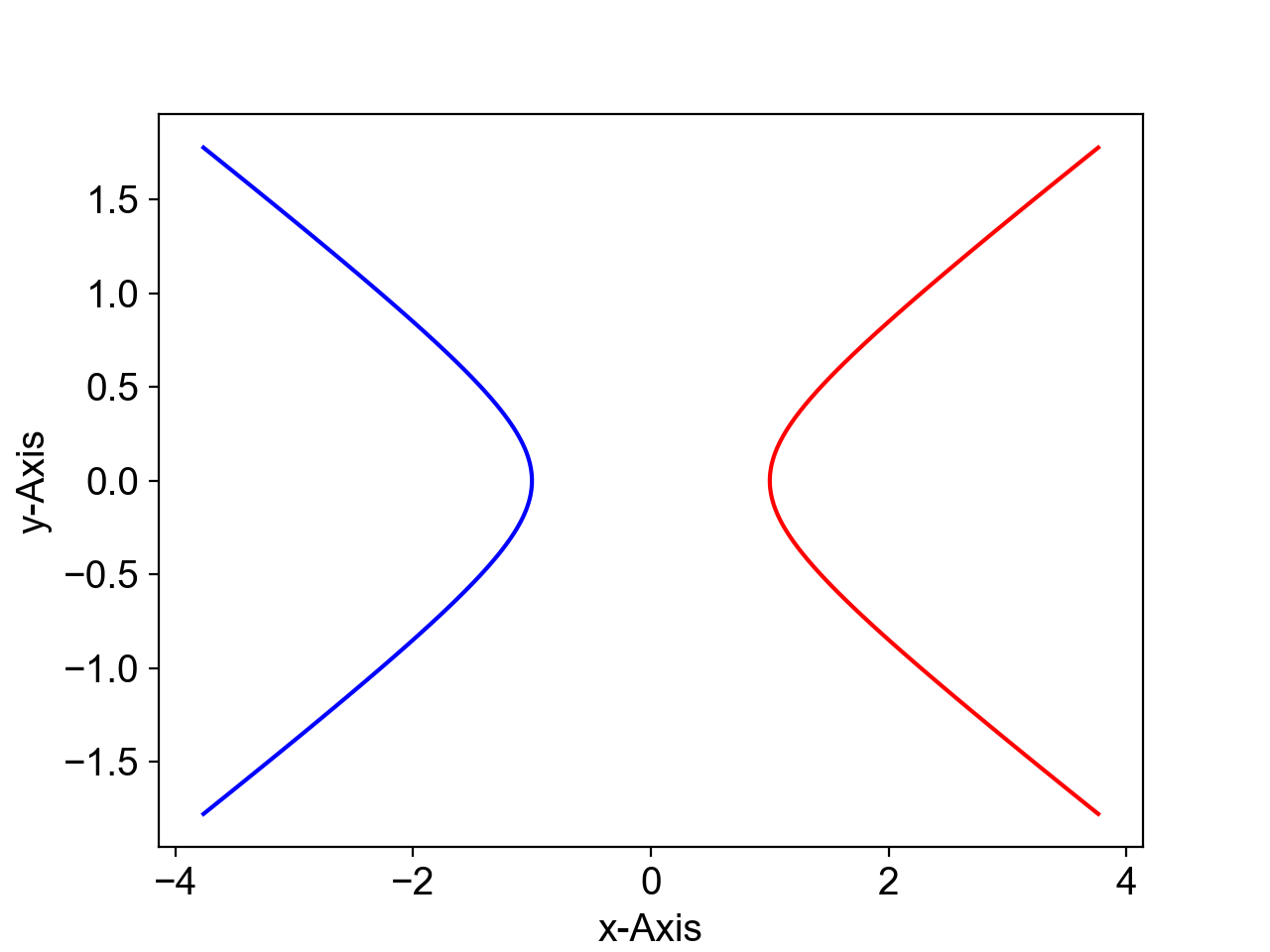
*y = b \* np.sinh(t)*

Jetzt bleibt uns nur noch übrig die Funktionen zu plotten. Wir definieren zwei Funktionen für das 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")*

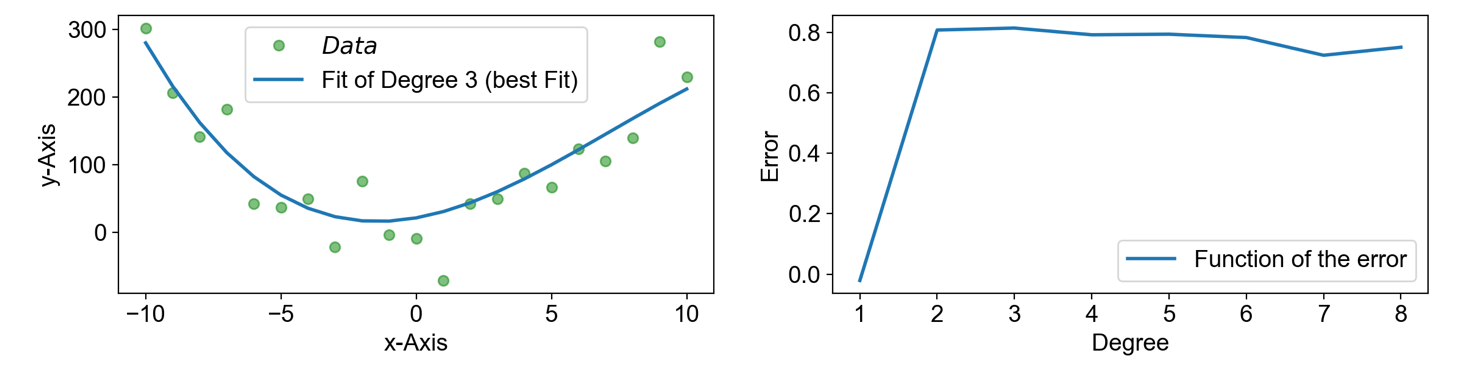
plt.show()



## 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):

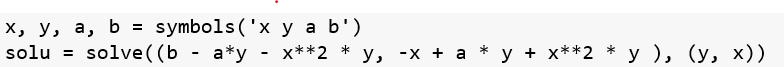
Dabei wird der Fehler zwischen Daten und Vorhersage berechnet und quadriert. Der Fehler bewegt sich damit zwischen einem Wert von null und eins, wobei eins als optimal und null als schlechtester Fit zu bewerten ist. Da wir unseren Fit nur auf Basis jedes zweiten Datenpunktes erstellt haben, können wir 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**

## c) 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:



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:

(a+x)\*\*2?

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

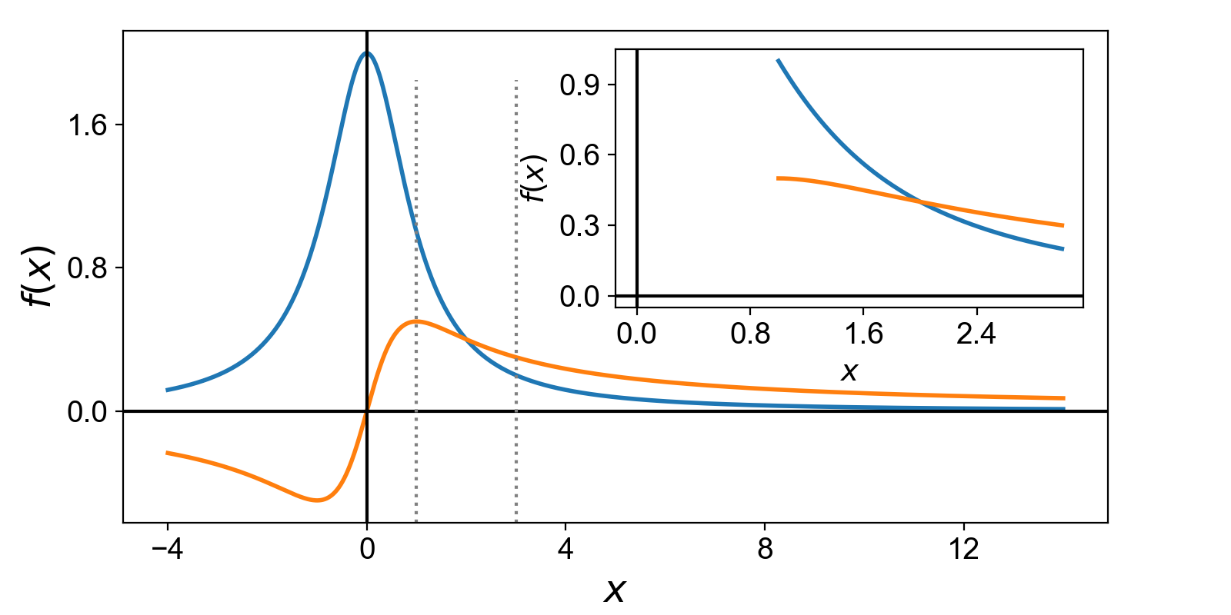
Bei einsetzen von y kürzt sich und es bleibt .

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:



Dies liefert das Ergebnis:

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

Da der Abstand zwischen Erde und Mond aber nur 384.400 ist und der Lagrange Punkt L1 zwischen Erde und Mond liegt kommt nur die Lösung in Betracht, da der andere Punkt sich hinter dem Mond befindet.