Iterative Verfahren

Klas Benjamin, Knoll Alexander

29. Mai 2016

Euler-Bernoulli-Balken

Der Euler-Bernoulli-Balken ist ein einfaches Modell für eine Biegevorgang auf Grund von Spannung. Bezeichnet $y(x)$ für $0 \leq x \leq L$ die vertikale Auslenkung, so gilt $EIy(x) = f(x)$ wobei E eine Material-Konstante und I das Trägheitsmoment ist. $f(x)$ beschreibt als Kraft pro Einheitslänge die Beladung des Balkens. Durch Diskretisierung erhält man aus der Differentialgleichung ein lineares Gleichungsystem, das hier iterativ gelöst werden soll.
Betrachtet wird dabei ein Stahlträger der Länge $L = 10m$ mit Tiefe $d = 5cm$ und Breite $b = 10cm$. Die Dichte von Stahl ist ungefähr $7850\frac{kg}{m^3}$, $E = 2 \cdot 10^1 1\frac{N}{m^2}$, $I = \frac{bd^3}{12}$

1 Unbelasteter Balken

Zunächst soll ein an beiden Seiten aufliegender Balken untersucht werden. Somit gilt $y(0) = y'(x0) = y(L) = y'(L) = 0$.
Um entscheiden zu können welches Verfahren zur Lösung des Problems geeignet ist, müssen diese auf Konvergenz untersucht werden. Das Problem ist gegeben durch

$$Ax = \frac{h^4}{EI} f, \tag{1}$$

wobei $h = \frac{L}{n+1}$ und $f = g \cdot f(x)$ gilt. Um zu prüfen ob eines der Verfahren konvergiert muss A auf gewisse eigenschaften besitzen. Die Verfahren Konvergieren wenn eine der folgenden bedingungen erfüllt sind.

- Jacobi Verfahren
  - Diagonaldominanz
- Gauß-Seidel Verfahren
  - Diagonaldominanz
    - Positiv-definit

1.1 Konvergenzuntersuchung

1.1.1 Diagonaldominanz

Beide Verfahren konvergieren sobald A diagonaldominant ist. Speziel für die gegebene Matrix lässt sich sagen, dass dies nicht der Fall ist. Für Strikte Diagonaldominanz muss gelten

$$\sum_{j=1, j\neq i}^{n} |a_{ij}| < |a_{ii}|,$$

bzw. für Schwache Diagonaldominanz

$$\sum_{j=1, j\neq i}^{n} |a_{ij}| \leq |a_{ii}|.$$

Somit wird ersichtlich dass aufgrund der beiden ersten und leitzen Zeilen A nicht Diagonaldominant ist und deswegen dass Jacobi Verfahren, in diesem Fall, nicht konvergiert.

1.1.2 Positiv Definititheit

Eine Matrix M ist positiv Definit genau dann wenn alle Eigenwerte größer als 0 sind. Zur bestimmung der Eigenwerte wird folgende Gleichung gelöst

$$det(A - E\lambda) = 0.$$

Glücklicherweise kürzen sich viele Terme da auf den meisten Diagonalen Null Einträge enthalten sind. Somit erzeugt einzig die Hauptdiagonale einen Term, für beliebig große Matrizen dieser Form, der dargestellt werden kann als

$$(12 - \lambda) \cdot \prod_{i=1}^{n-1} (6 - \lambda) \cdot (12 - \lambda) = 0$$

Somit sind nur die Eigenwerte $\lambda = 12$ und $\lambda = 6$ lösungen. Da diese Eigenwerte positiv sind folgt, dass das Gaus-Seidel Verfahren für beliebig große Matrizen A konvergiert.

1.2 Implementierung Gauss-Seidel

Beim Gauss-Seidel verfahren werden immer die Aktuellsten Werte der Lösung verwendet um einen Wert für die Nächste Stelle zu berechnen. Hierfür wird Gleichung (1) nach dem jeweiligen x aufgelöst. Somit erhält man die Form

$$x_k^{(m+1)} := \frac{1}{a_{ii}} \left(b_k - \sum_{i=1}^{k-1} a_{ki} \cdot x_i^{(m+1)} - \sum_{i=k+1}^{n} a_{ki} \cdot x_i^m \right), \tag{2}$$

wobei b_k die rechte Seite der von Gleichung (1) ist, und a die jeweiligen einträge der Matrix A. Eine Implementierungsmöglichkeit wäre es Bandmatrizen zu benutzen, da diese wegen aussparung der Nullen in diesem Fall einen deutlich geringeren Speicherbedarf besitzen. Jedoch für dass Speziell gegebene Problem ist es sinnvoller hart Kodierte Terme zu verwenden. Somit fällt nähmlich die Zugriffszeit auf Elemente, als auch der komplette Speicherbedarf, weg. Eine mögliche Umsetzung könnte wie folgt aussehen

```
1     def inner_sum(ys, index):
2         if 1 < index < len(ys)-2:
3             result = ys[index-2] - 4 * ys[index-1] - 4 * ys[index+1] + ys[index+2]
4         elif index == 0:
5             result = -6 * ys[index+1] + (4.0/3.0) * ys[index+2]
6         elif index == 1:
7             result = -4 * ys[index-1] - 4 * ys[index+1] + ys[index+2]
8         elif index == len(ys)-2:
9             result = ys[index-2] - 4 * ys[index-1] - 4 * ys[index+1]
10        else:
11            result = (4.0/3.0) * ys[index-2] - 6 * ys[index-1]
12        return result
13
14    def seidel(ys, bs):
15      ys_old = list(ys) # anlegen einer kopie, somit gehen alte Werte nicht verloren
16        for i in range(len(ys)):
17          diag_elem = 6.0
18            if i == 0 or i == len(ys)-1:
19              diag_elem = 12.0
20            ys[i] = (bs[i] - inner_sum(ys, i))/diag_elem
21        return ys, ys_old
```

Die Funktion *seidel* nimmt als argument zwei Listen. Die aktuelle Lsung x^m und die rechte Seite der Gleichung b. Da der Algorithmus die bergebene Liste der aktuellen Lsung bearbeitet, also ihren Zustand verndert, ist es notwendig sie vorher in einer Kopie zu sichern. Somit gehen die alten Werte nicht verloren und es knnen weitere Untersuchungen vorgenommen werden. Die Funktion *inner_sum*, aufgerufen durch *seidel* berechnet ausgehend von einem Index, wie der Name es schon vermuten lsst, die innere Summen von Gleichung (2), also

$$inner\_sum = \sum_{i=1}^{k-1} a_{ki} \cdot x_i^{(m+1)} + \sum_{i=k+1}^{n} a_{ki} \cdot x_i^m.$$

Dieser Algorithmus wird iterativ von der Funktion *solve* aufgerufen. Parameter fr *solve* sind der Startvektor x^0, die rechte Seite b, die Anzahl an maximal Iterationen als auch eine Toleranz fr den Lsungsfortschritt.

```
1     def solve(ys, bs, max_iter=40000, tol=1e-6):
2         it_counter = 0
3         iterating = True
4         while iterating:
5             _, ys_old = seidel2(ys, bs)
6             step = max([a_i - b_i for a_i, b_i in zip(ys, ys_old)])
7             it_counter += 1
8             if it_counter >= max_iter or abs(step) <= tol:
9                 iterating = False
10        return it_counter
```

Die Funktion iteriert solange bis entweder die Anzahl an maximal iterationen erreicht, oder bis dass Kriterium

$$\max[x^{m+1} - x^m] < tol$$

erfllt ist.

1.2.1 Parallelisierung

Der Algorithmus selbst ist nicht parallelisierbar, da fÃr die aktuelle Berechnung immer die aktuellsten Werte benÃtigt werden. Dass macht es unmÃglich die dass Problem aufzuteilen und von verschiedenen Prozessoren bearbeiten zu lassen. Was jedoch mÃglich ist, ist die bearbeitung des Problems fÃr verschieden groÃe Gleichungssysteme. Wenn gilt

$$n = 10 \cdot 2^k + 1$$

fÃr $k = 1, .., 10$. Es ist also mÃglich fÃr jedes k dass Problem separat, und somit parallel, zu bearbeiten. HierfÃr kÃnnen verschiedene Strategien verwendet werden. In diesem Fall wurde Pythons *threading* Packet verwendet.

```
def multi_solve(k_limit, max_iter=10000, tol=1e-10):
  start = time.time()
  # listen initialisieren
    n_list = [10 * (2**k) + 1 for k in range(1, k_limit+1, 1)]
    thread_list = [None for _ in n_list]
    line_space_list = [[] for _ in n_list]
    ys_list = [0 for _ in n_list]
    for n in n_list:
      # werte f r solve initialisieren
      index = n_list.index(n)
        h = length/(n+1)
        ys_list[index] = [0 for _ in range(n)]
        fs = [f() for _ in range(n)]
        bs = list(map(lambda x: x * ((h**4) / (E*I)), fs))
        line_space_list[index] = np.linspace(0, length, n+2)
        # solve f r aktuelle werte in neuem thread starten
        thread_list[index] = thr.Thread(target=solve,
                        args=(ys_list[index], bs, max_iter, tol))
        thread_list[index].start()

    for thread in thread_list:
        index = thread_list.index(thread)
        # warten bis thread fertig ist
        thread.join()
        ys_list[index] = [0] + ys_list[index] + [0]
        plt.plot(line_space_list[index], [-x for x in ys_list[index]])
        error , error_index = compute_error(ys_list[index],
                                  compute_exact(line_space_list[index]))
        print('Max error: %s for k=%s: at index: %s of %s'%(error,
                                index+1, error_index, len(ys_list[index])-1))
  end = time.time()
    print("time needed in seconds: ", end - start)
    plt.show()
```

Die Funktion *multi_solve* nimmt als Parameter ein oberes Limit fÃr k als auch wieder eine obere Grenze fÃr die Anzahl der maximal Iterationen und eine Toleranz. FÃr jedes k wird dabei ein neuer Thread gestartet. AnschlieÃend wird durch *thread.join* auf die beendigung eines jeden Trheads gewartet. Allerdings hat diese Implementierung den Nachteil dass nicht darauf geachtet wird ob alle VerfÃgbaren Resourcen der Prozessoren genutz werden. Im Schnitt liegt die Auslastung auf einem Ãblichen Rechner bei 30% bis 40%. AuÃerdem sollte beachtet werden dass der Speicherbedarf grÃÃer wird. WÃrde man statt Hart Kodierten Termen fÃr die Berechnung der LÃsung, normale Matrizen verwenden, so wÃrde dass bei einem 4GB RAM Rechner zu Problemen fÃhren.

1.3 Auswertung

In diesem Abschnitt sollen die numerische Ergebnisse untersucht und nach mÃglichkeit weiter begrÃndet werden. FÃr den Vergleich wird die Analytische LÃsung des Problems einbezogen, welche gegeben ist durch

$$y(x) = \frac{fx^2(L-x)^2}{24EI} \tag{3}$$

1.3.1 Auswertung n = 10

ZunÃchst soll dass Gleichungssystem einzeln fÃr $n = 10$ gelÃst werden. DafÃr wird die oben definierte Funktion *solve* mit den iterations Parametern $max\_iter = 1000000$ und $tol = 1e^{-6}$, fÃr 6-Stellige genauigkeit, aufgerufen.

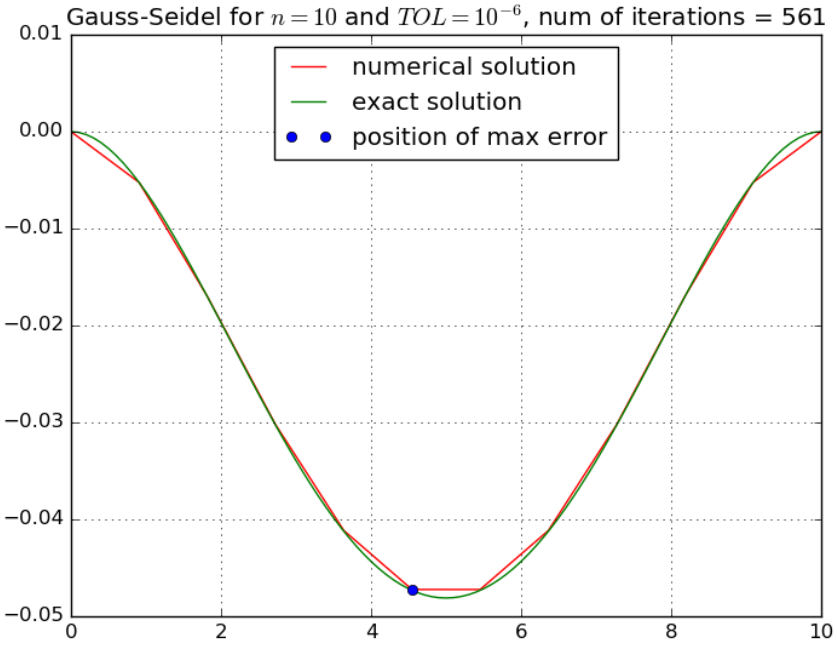

Abbildung 1: Vergleich numerische und analytische LÃsung

FÃr eine 6-Stellige Genauigkeit werden 561 iterationen benÃtigt. Dabei betrÃgt der Fehler, welcher in der Mitte des Balkens liegt, ca. $8.75913330772e^{-5}$.

1.3.2 Auswertung fÃr k's

Nun werden die Gleichungssysteme mittels *multi_solve* gelÃst. ZunÃchst werden dabei die iterations Parameter $max\_iter = 10000$ und $tol = 1e^{-6}$ verwendet. Die Auswertung liefert folgende Fehlertabelle fÃr den Fehler in der Mitte des Balkens. Man kann erkennen

k	Fehler
1	4.0104791661e-05
2	0.0285147270488
3	0.0467199915587
4	0.0480244275509
5	0.0481081478527
6	0.0481135140356

dass bereits fÃr relativ kleine k der Fehler recht groÃ wird. Bereits ab $k = 3$ ist dass Resultat fÃr diegenutzen Parameter unbrauchbar. Der Grund hierfÃr liegt darin dass fÃr

groÃe n dass Verfahren nur sehr langsam konvergiert. Der Plot fÃr $k = 1..10$ verdeutlicht die Situation.

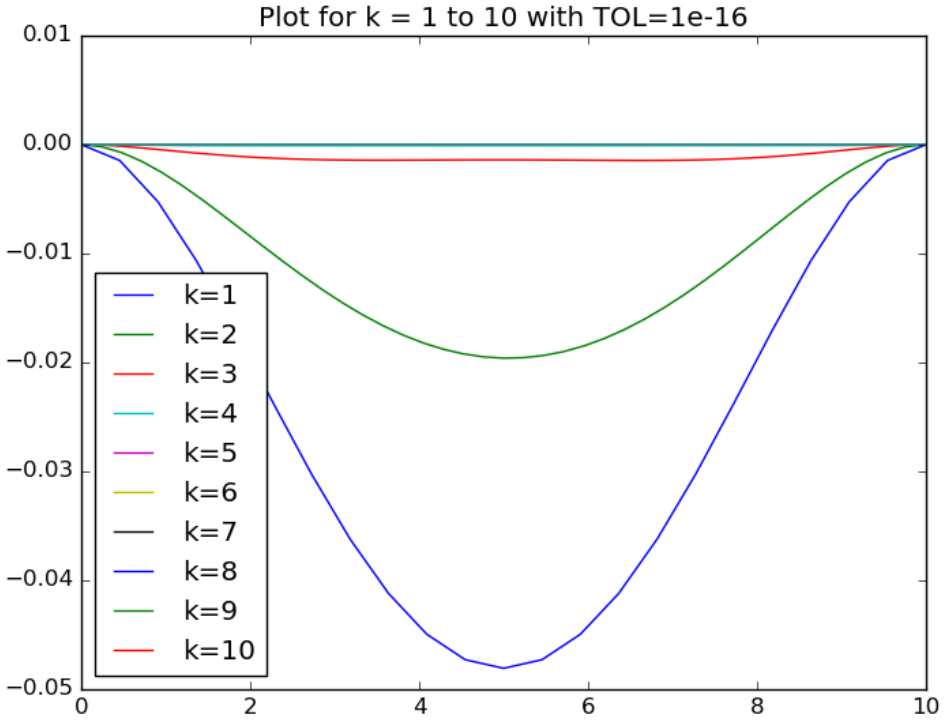

Abbildung 2: Plots fÃr k=1..10

Bereits ab $k = 2$ kann man im Plot sehen dass das Resultat nicht zufriedenstellend ist, da der Fehler viel zu groÃ wird. Abhilfe verschaffen hierfÃr besser geeignete iterations Parameter. Die folgende Auswertung verwendet die Parameter $max\_iter = 1000000$ und $TOL = 1e^{-100}$. Die Rechenzeit betrug dabei $t \approx 8h$

k	**Fehler**
1	1.27883814649e-14
2	1.7937734631e-13
3	0.00121666321593
4	0.0382262076262
5	0.0475469272666
6	0.0480776249649

Es zeigt sich dass die NÃherungen besser werden, jedoch bereits ab $k = 4$ die Anzahl an iterationen mit $max\_iter = 1000000$ zu niedrig angesetzt ist. Der Plot verdeutlicht dass selbst nach so langer rechenzeit, nicht merklich besser werden.

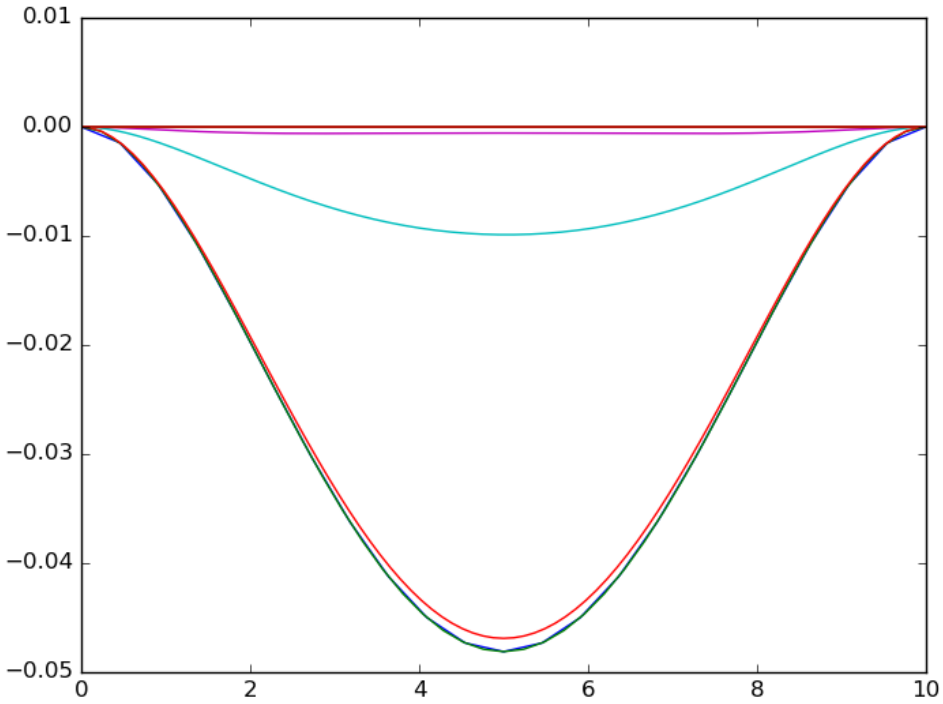

Abbildung 3: Plots mit max_iter = 1000000

Bereits fÃr kleine k wird der Fehler in der Mitte des Balkens sehr schnell, sehr groÃ.

1.3.3 Fehlerbetrachtung

Die Kondition sagt etwas darÃber aus ob kleine stÃrungen zu verstÃrkung des Fehlers Ãber alle Grenzen fÃhren. Wenn fÃr den Konditionswert gilt $K(A) < 1$, so spricht man von von eime gut Konditionierten (gut gestelltem) Problem. FÃr regulÃre Matrizen lÃsst sich der Konditionswert abschÃtzen durch

$$K(A) = ||A|| \cdot ||A^{-1}||. \tag{4}$$

HierfÃr muss jedoch gelten dass fÃr beliebig groÃe Gleichungssysteme die Matrix A regulÃr ist (also vollen rang hat). Dies gilt genau dann wenn

$$det(A) \neq 0.$$

BegrÃnden lÃsst sich dies wie bereits bei der Konvergenzuntersuchung. Da alle diagonalen, bis auf die Hauptdiagonale mindestens ein Null Element enthalten, kann die Determinante wie folgt dargestellt werden

$$12 \cdot \prod_{i=1}^{n-1} 6 \cdot 12.$$

Dieser Ausdruck ist immer grÃÃer als Null, somit ist A fÃr beliebige grÃÃen regulÃr. Also kann Gleichung (4) zur bestimmung der Kondition verwendet werden.

1.3.4 Konditionsbestimmung

Die Kondition des Problems fÃr verschiedene k soll mithilfe von Python bestimmt werden. HierfÃr werden die Funktionen *norm* und *inv* aus dem Paket *numpy.linalg* verwendet. Notwendigerweise werden nun die Matrizen tatsÃchlich konstruiert, dies geschieht mittels der Funktion *create_matrix* welches als Parameter die Dimension n erwartet.

```
1  def create_matrix(n):
2      matrix = [[] for _ in range(n)]
3      matrix[0] = [12, -6, 4.0/3.0] + [0 for _ in range(n-3)]
4      matrix[1] = [-4, 6, -4, 1] + [0 for _ in range(n-4)]
5      matrix[n-2] = [0 for _ in range(n-4)] + [1, -4, 6, -4]
6      matrix[n-1] = [0 for _ in range(n-3)] + [4.0/3.0, -6, 12]
7      for j in range(2, n-2, 1):
8        matrix[j] = [0 for _ in range(j-2)] + [1, -4,  6, -4, 1] + [0 for _ in range(n-3-j)]
9      return matrix
```

Die so erzeugten Matrizen werden verwendet um mittels Gleichung (4) den Konditionswert fÃr dass jeweilige Gleichungssystem zu bestimmen. Zur berechnung der Konditionswerte wird die Funktion *compute_conditions* verwendet, welche ein oberes Limit fÃr den Wert k bekommt. Als Resultat wird eine Liste mit den entsprechhenden Konditionswerten zurÃckgeliefert.

```
1          def compute_conditions(k_limit):
2              n_list = [10 * (2**k) + 1 for k in range(1, k_limit+1, 1)]
3              cond_list = [0 for _ in n_list]
4              for n in n_list:
5                  index = n_list.index(n)
6                  matrix = create_matrix(n)
7                  inverse_matrix = nplin.inv(matrix)
8                  matrix_norm = nplin.norm(matrix)
9                  inverse_matrix_norm = nplin.norm(inverse_matrix)
10                 cond_list[index] = matrix_norm*inverse_matrix_norm
11             return cond_list
```

Die Funktion bildet zu jeder Matrix die Inverse und bestimmt die jeweiligen Normen.

Ein Plot der die Kondition zu jedem Wert fÃr k darstellt, verdeutlicht dass verhalten.

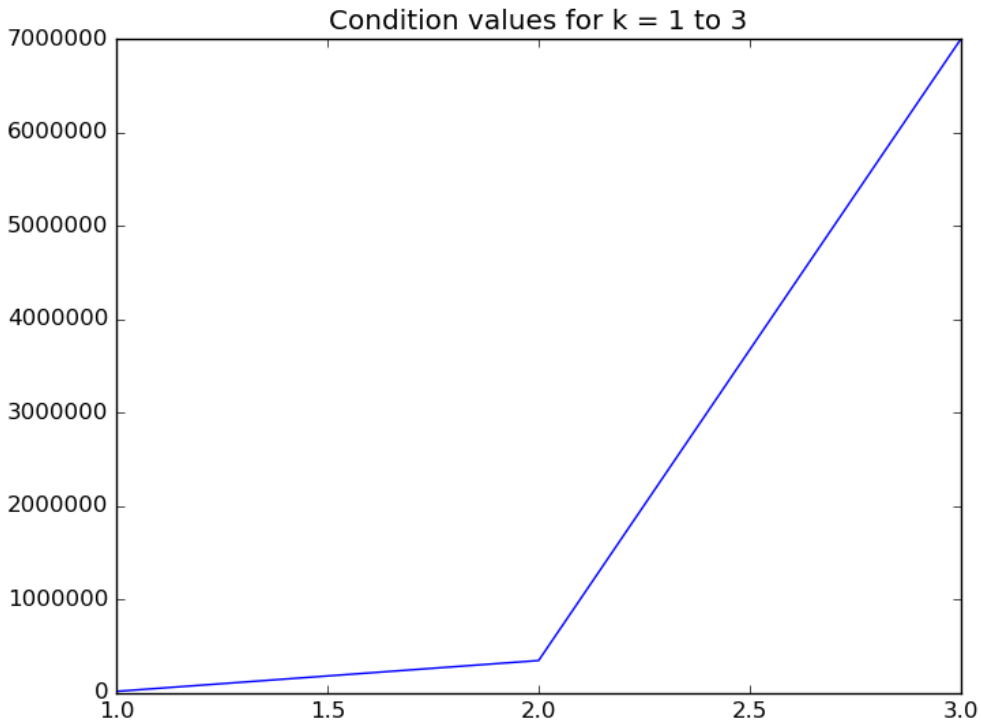

Abbildung 4: Plots mit Konditionen

2 Belasteter Balken

Nun soll der aufliegende Balken zwischen $x = 3$ und $x = 4$ zusÃtzlich mit $500kg$ belastet werden. Da die angreifende Kraft bereits als eine Liste in den algorithmen implementiert wurde ist es bereits aussreichend die rechte seite vor dem AusfÃhren der Routinen anzupassen. HierfÃr wird die Funcktion *apply_weight_to_f* verwended. Diese erhÃlt als Argumente die Liste mit den standartmÃÃig wirkenden KrÃften fs, als auch eine Schrittweite h. FÃr jedes Element dessen tatsÃchliche Position zwischen $x = 3$ und $x = 4$ liegt wird die ZusÃtzliche belastung von $500kg$ dazu addiert.

```
1    def apply_weight_to_f(fs, h):
2        for i in range(len(fs)):
3            if 3 <= h*i <= 4:
4                fs[i] += 500
```

Die Funktion hat keinen RÃckgabewert da sie den Zustand der Liste selbst verÃndert.

2.1 Auswertung

Da nun keine exakte LÃsung vorhanden ist an der ein Fehler gemessen werden kann, wird in diesem Teil darauf Verzichtet. Stattdessen wird die verschiebung der maximalen

Auslenkung untersucht. HierfÃr wird die Routine *multi_solve* mit angepasster rechter Seite ausgefÃhrt. Diesmal werden jedoch die Parameter $max_iter = 100000$ und $tol = 1e^{-18}$ verwendet.

2.1.1 Auswertung n = 10

Der Vergleich der beiden Numerischen lÃsungen, einmal mit angepasster Seite und einmal ohne, liefert folgenden Plot.

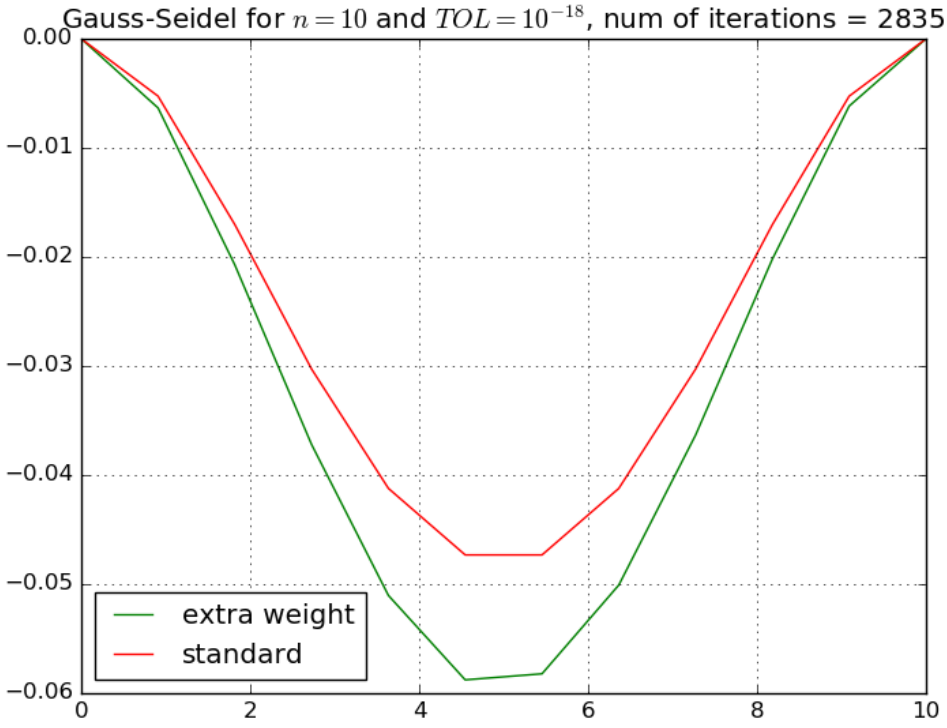

Abbildung 5: Plots mit max_iter = 1000000

Es ist erkenntlich, dass der Punkt der grÃÃten Auslenkung fÃr $n = 10$ sich nicht verÃndert hatt, jedoch der Betrag der Auslenkung gewachsen ist. Deutlicher wird die linksverschiebung der grÃÃten Auslenkung fÃr grÃÃere n.

2.1.2 Auswertung fÃr k's

Dieses mal werden nur $k <= 5$ ausgewertet um die berechnungen in einem Zeitlichen Rahmen halten zu kÃnnen. FÃr die Konvergenzgeschwindigkeit spielt die rechte Seite, in diesem Fall, eine untergeordnete Rolle. Somit ist nicht zu erwarten dass die Resultate diesmal besser, oder schneller sind. Die berechnungen werden mit einer angepassten *multi_solve* Funktion durchgefÃhrt (nur anpassung bezÃglich der Ausgabe, keine Algorithmischen).

3 Herleitung der Diskretisierung

4 Eingespannter Balken

5 Vergleich der Tragfhigkeit

6 SOR-Verfahren

7 cg-Verfahren

Literatur