

Iterative Verfahren

Klas Benjamin, Knoll Alexander

29. Mai 2016

Euler-Bernoulli-Balken

Der Euler-Bernoulli-Balken ist ein einfaches Modell für einen 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 Gleichungssystem, 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 $7850 \frac{kg}{m^3}$, $E = 2 \cdot 10^{11} \frac{N}{m^2}$, $I = \frac{bd^3}{12}$

1 Unbelasteter Balken

Zunächst soll ein an beiden Seiten aufliegender Balken untersucht werden. Somit gilt $y(0) = y'(0) = y(L) = y'(L) = 0$.

Um entscheiden zu können welches Verfahren zur Lösung des Problems geeignet ist, müssen diese auf Konvergenz untersucht werden. Das Problem ist gegeben durch

$$Ax = \frac{h^4}{EI} f, \quad (1)$$

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. Speziell 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 letzten Zeilen A nicht Diagonaldominant ist und deswegen dass Jacobi Verfahren, in diesem Fall, nicht konvergiert.

1.1.2 Positiv Definitheit

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 Gauss-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

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

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 nämlich 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

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

erflft ist.

1.2.1 Parallelisierung

Der Algorithmus selbst ist nicht parallelisierbar, da fÄr die aktuelle Berechnung immer die aktuellsten Werte benÄtigt werden. Dass macht es unmÄglich die dass Problem aufzuteilen und von verschiedenen Prozessoren bearbeiten zu lassen. Was jedoch mÄglich ist, ist die bearbeitung des Problems fÄr verschieden groÄe Gleichungssysteme. Wenn gilt

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

fÄr $k = 1, \dots, 10$. Es ist also mÄglich fÄr jedes k dass Problem separat, und somit parallel, zu bearbeiten. HierfÄr kÄnnen verschiedene Strategien verwendet werden. In diesem Fall wurde Pythons *threading* Packet verwendet.

```

1 def multi_solve(k_limit, max_iter=10000, tol=1e-10):
2     start = time.time()
3     # listen initialisieren
4     n_list = [10 * (2**k) + 1 for k in range(1, k_limit+1, 1)]
5     thread_list = [None for _ in n_list]
6     line_space_list = [[] for _ in n_list]
7     ys_list = [0 for _ in n_list]
8     for n in n_list:
9         # werte f r solve initialisieren
10        index = n_list.index(n)
11        h = length/(n+1)
12        ys_list[index] = [0 for _ in range(n)]
13        fs = [f() for _ in range(n)]
14        bs = list(map(lambda x: x * ((h**4) / (E*I)), fs))
15        line_space_list[index] = np.linspace(0, length, n+2)
16        # solve f r aktuelle werte in neuem thread starten
17        thread_list[index] = thr.Thread(target=solve,
18                                       args=(ys_list[index], bs, max_iter, tol))
19        thread_list[index].start()
20
21    for thread in thread_list:
22        index = thread_list.index(thread)
23        # warten bis thread fertig ist
24        thread.join()
25        ys_list[index] = [0] + ys_list[index] + [0]
26        plt.plot(line_space_list[index], [-x for x in ys_list[index]])
27        error, error_index = compute_error(ys_list[index],
28                                           compute_exact(line_space_list[index]))
29        print('Max error: %s for k=%s: at index: %s of %s'%(error,
30                                                           index+1, error_index, len(ys_list[index])-1))
31    end = time.time()
32    print("time needed in seconds: ", end - start)
33    plt.show()

```

Die Funktion *multi_solve* nimmt als Parameter ein oberes Limit für k als auch wieder eine obere Grenze für die Anzahl der maximal Iterationen und eine Toleranz. Für jedes k wird dabei ein neuer Thread gestartet. Anschließend wird durch *thread.join* auf die Beendigung eines jeden Threads gewartet. Allerdings hat diese Implementierung den Nachteil, dass nicht darauf geachtet wird, ob alle verfügbaren Ressourcen der Prozessoren genutzt werden. Im Schnitt liegt die Auslastung auf einem üblichen Rechner bei 30% bis 40%. Außerdem sollte beachtet werden, dass der Speicherbedarf größer wird. Würde man statt hart kodierten Termen für die Berechnung der Lösung, normale Matrizen verwenden, so würde das bei einem 4GB RAM Rechner zu Problemen führen.

1.3 Auswertung

In diesem Abschnitt sollen die numerischen Ergebnisse untersucht und nach Möglichkeit weiter begründet werden. Für den Vergleich wird die analytische Lösung des Problems einbezogen, welche gegeben ist durch

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

1.3.1 Auswertung $n = 10$

Zunächst soll das Gleichungssystem einzeln für $n = 10$ gelöst werden. Dafür wird die oben definierte Funktion *solve* mit den Iterationsparametern $max_iter = 1000000$ und $tol = 1e^{-6}$, für 6-stellige Genauigkeit, aufgerufen.

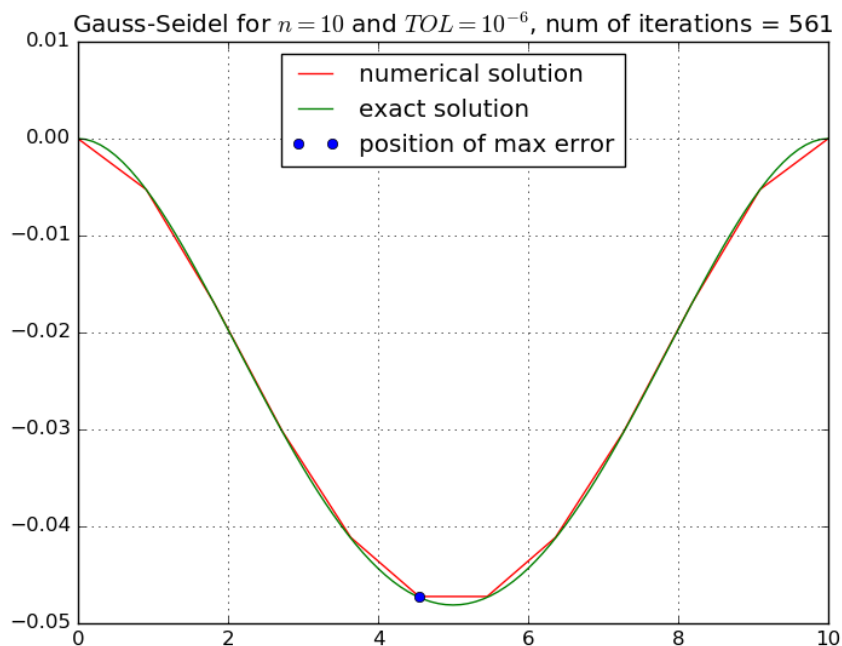


Abbildung 1: Vergleich numerische und analytische Lösung

Für eine 6-stellige Genauigkeit werden 561 Iterationen benötigt. Dabei beträgt der Fehler, welcher in der Mitte des Balkens liegt, ca. $8.75913330772e^{-5}$.

1.3.2 Auswertung für k 's

Nun werden die Gleichungssysteme mittels *multi_solve* gelöst. Zunächst werden dabei die Iterationsparameter $max_iter = 10000$ und $tol = 1e^{-6}$ verwendet. Die Auswertung liefert folgende Fehlertabelle für 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 für relativ kleine k der Fehler recht groß wird. Bereits ab $k = 3$ ist das Resultat für die genannten Parameter unbrauchbar. Der Grund hierfür liegt darin, dass für

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

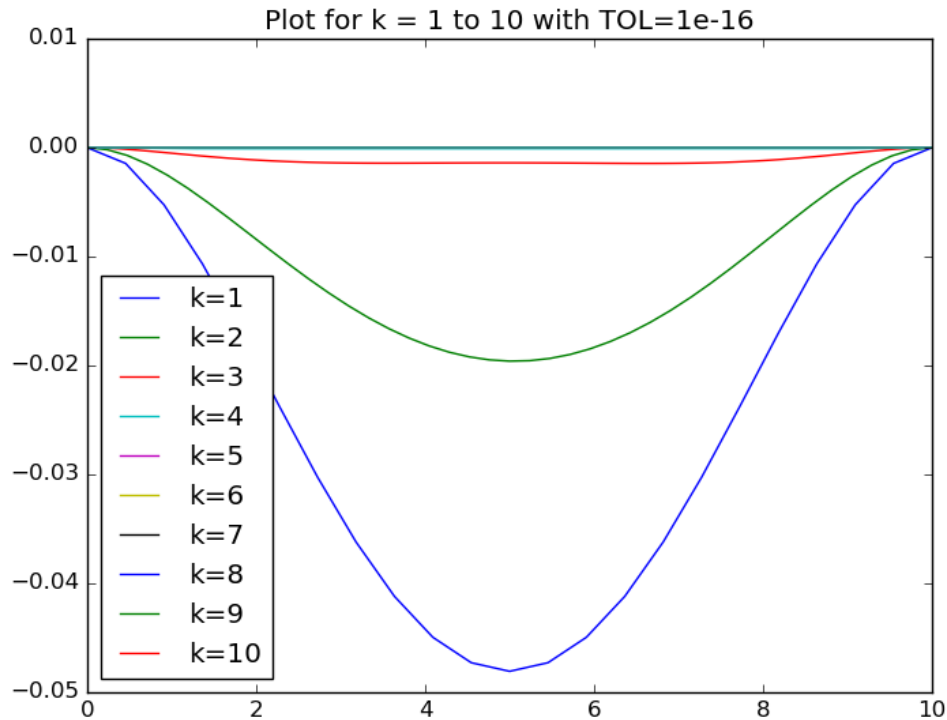


Abbildung 2: Plots für $k=1..10$

Bereits ab $k = 2$ kann man im Plot sehen dass das Resultat nicht zufriedenstellend ist, da der Fehler viel zu groß wird. Abhilfe verschaffen hierfür 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 Näherungen 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.

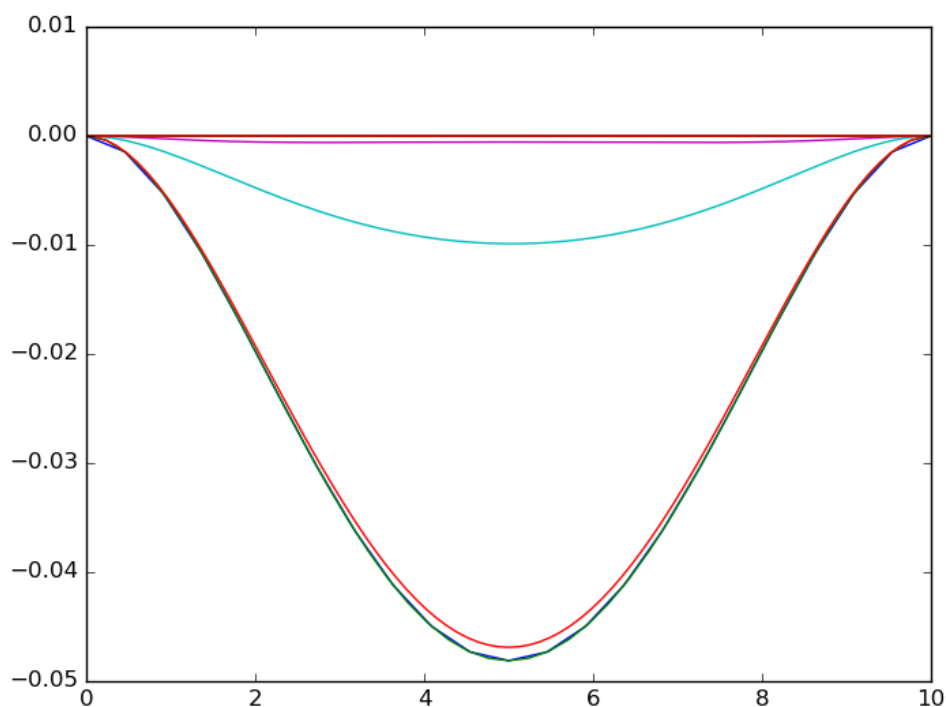


Abbildung 3: Plots mit $max_iter = 1000000$

Bereits für kleine k wird der Fehler in der Mitte des Balkens sehr schnell, sehr groß.

1.3.3 Fehlerbetrachtung

Die Kondition sagt etwas darüber aus ob kleine Störungen zu Verstärkung des Fehlers über alle Grenzen führen. Wenn für den Konditionswert gilt $K(A) < 1$, so spricht man von einem gut konditionierten (gut gestelltem) Problem. Für reguläre Matrizen lässt sich der Konditionswert abschätzen durch

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

Hierfür muss jedoch gelten dass für beliebig große Gleichungssysteme die Matrix A regulär ist (also vollen Rang hat). Dies gilt genau dann wenn

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

Begründen lässt 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 größer als Null, somit ist A für beliebige n regulär. Also kann Gleichung (4) zur Bestimmung der Kondition verwendet werden.

1.3.4 Konditionsbestimmung

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

```

1 def create_matrix(n):
2     matrix = [[0 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 für dass jeweilige Gleichungssystem zu bestimmen. Zur Berechnung der Konditionswerte wird die Funktion *compute_conditions* verwendet, welche ein oberes Limit für den Wert k bekommt. Als Resultat wird eine Liste mit den entsprechenden Konditionswerten zurückgeliefert.

```

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 f für k darstellt, verdeutlicht das Verhalten.

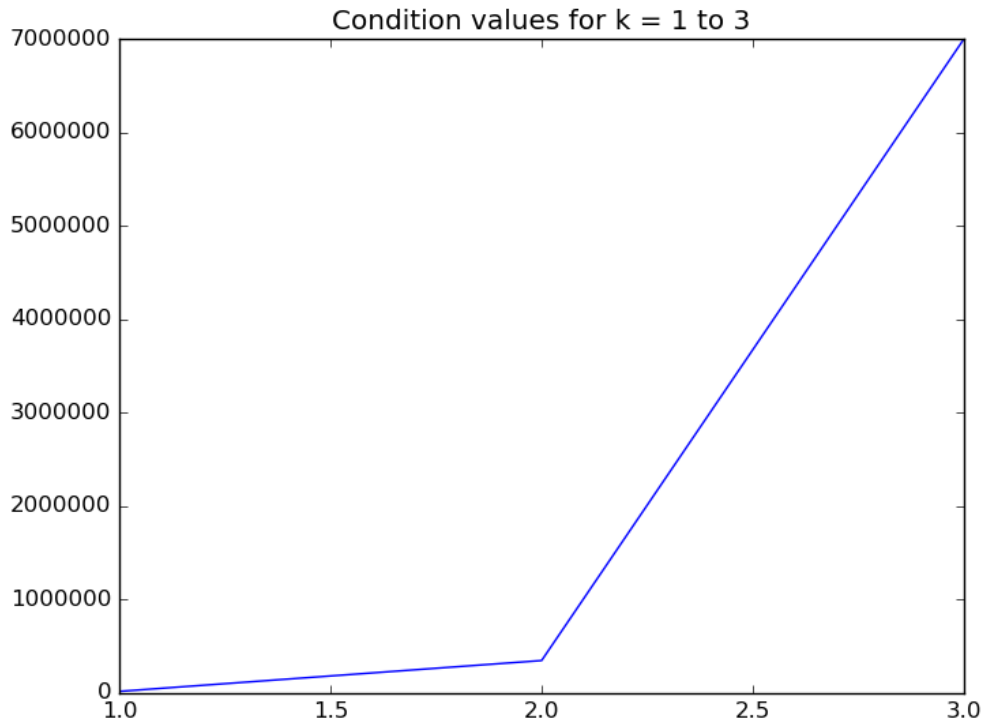


Abbildung 4: Plots mit Konditionen

2 Belasteter Balken

Nun soll der aufliegende Balken zwischen $x = 3$ und $x = 4$ zusätzlich mit 500kg belastet werden. Da die angreifende Kraft bereits als eine Liste in den algorithmen implementiert wurde ist es bereits ausreichend die rechte Seite vor dem Ausführen der Routinen anzupassen. Hierfür wird die Funktion *apply_weight_to_f* verwendet. Diese erhält als Argumente die Liste mit den standardmäßig wirkenden Kräften fs , als auch eine Schrittweite h . Für jedes Element dessen tatsächliche Position zwischen $x = 3$ und $x = 4$ liegt wird die zusätzliche 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 Rückgabewert da sie den Zustand der Liste selbst verändert.

2.1 Auswertung

Da nun keine exakte Lösung vorhanden ist an der ein Fehler gemessen werden kann, wird in diesem Teil darauf verzichtet. Stattdessen wird die Verschiebung der maximalen

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

2.1.1 Auswertung $n = 10$

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

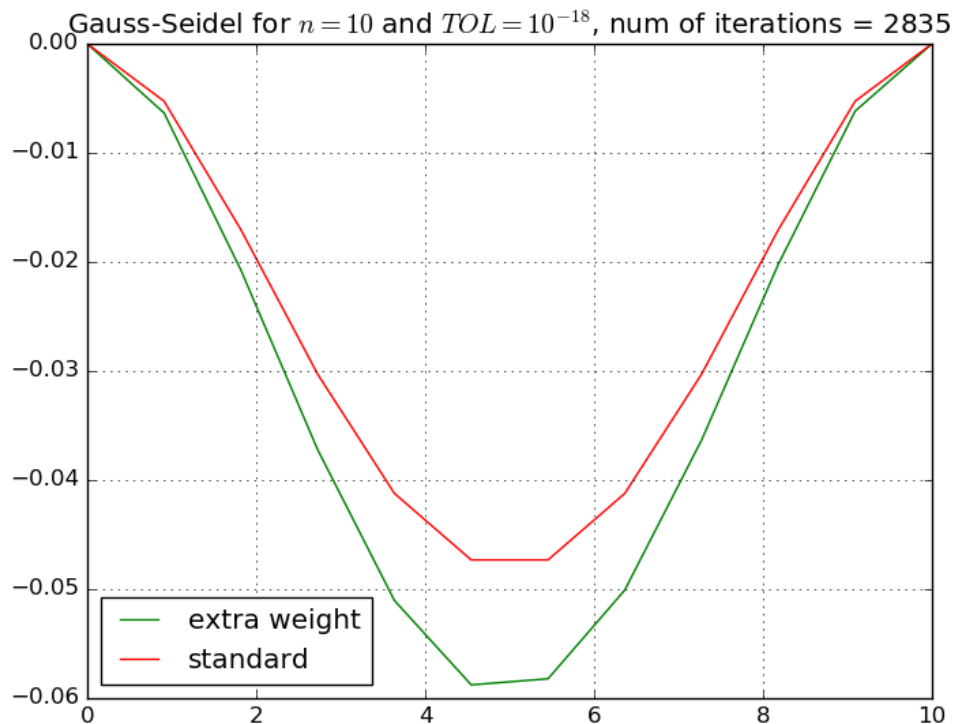


Abbildung 5: Plots mit $max_iter = 1000000$

Es ist erkenntlich, dass der Punkt der größten Auslenkung für $n = 10$ sich nicht verändert hat, jedoch der Betrag der Auslenkung gewachsen ist. Deutlicher wird die linksverschiebung der größten Auslenkung für größere n .

2.1.2 Auswertung für k 's

Dieses mal werden nur $k \leq 5$ ausgewertet um die Berechnungen in einem zeitlichen Rahmen halten zu können. Für die Konvergenzgeschwindigkeit spielt die rechte Seite, in diesem Fall, eine untergeordnete Rolle. Somit ist nicht zu erwarten, dass die Ergebnisse diesmal besser, oder schneller sind. Die Berechnungen werden mit einer angepassten *multi_solve* Funktion durchgeführt (nur Anpassung bezüglich der Ausgabe, keine algorithmischen).

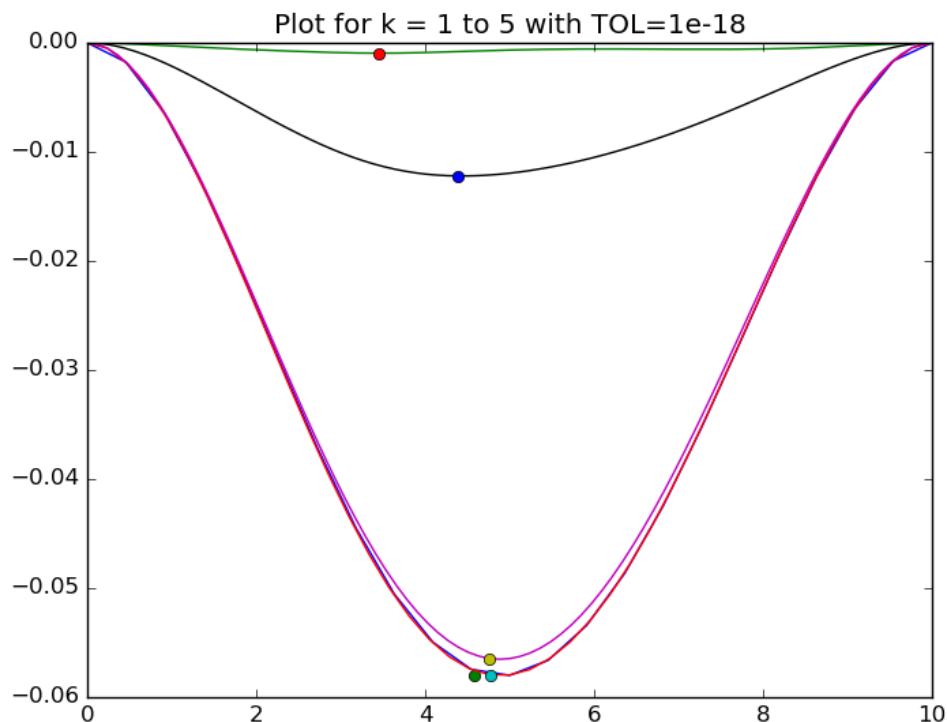


Abbildung 6: Plots mit $\text{max_iter} = 1000000$

Dem Plot kann entnommen werden, dass sich der Punkt mit maximaler Auslenkung tatsächlich leicht nach links verschiebt. Nämlich zwischen $x = 4$ und $x = 5$

- 3 Herleitung der Diskretisierung**
- 4 Eingespannter Balken**
- 5 Vergleich der Tragfähigkeit**
- 6 SOR-Verfahren**
- 7 cg-Verfahren**

Literatur