Numerische Mathematik - Projektteil 2

Richard Weiss Florian Schager Christian Sallinger Fabian Zehetgruber
Paul Winkler Christian Göth

Random code snippet, damit alle checken, wie man code displayt:

```
print("Hello World!")
```

1 Dünn besetzte Matrizen

Eine häufige Problemstellung in der Numerischen Mathematik lautet lineare Gleichungssysteme mit großen, dünn besetzten Matrizen zu lösen. Dabei kommen meist iterative Verfahren zum Einsatz, die in diesm Projekt effizient implementiert werden.

a)

Generieren Sie eine symmetrisch positiv definite Zufallsmatrix $A \in \mathbb{R}^{n \times n}$, wobei pro Zeile eine fixe Anzahl an Einträgen ungleich Null sind. Testen Sie für eine zufällige rechte Seite $b \in \mathbb{R}^n$ bis zu welcher Größe n das lineare Gleichungssystem

$$Ax = b$$

mit einem direkten Löser (numpy.linalg.solve) in akzeptabler Zeit gelöst werden kann. Welchen Aufwand erwarten Sie abhängig von n? Plotten Sie die Rechenzeit in Abhängigkeit von der Problemgröße.

```

def zufallsmatrix(n, nonzeros):
  A = np.concatenate((np.zeros((n,nonzeros)), np.random.rand(n, n-nonzeros)), axis = 1)
  for i in range(n):
    np.random.shuffle(A[i])
  return A + A.T + np.diag(np.random.rand(n)*10)

A_base = zufallsmatrix(5000,100)
```

Wie in Abbildung 1 ersichtlich verhält sich die Rechenzeit kubisch in Relation zur Problemgröße. Zum Testen wurde eine Zufallsmatrix mit 100 Nicht-Null-Einträgen pro Zeile (nicht exakt, da die Symmetriesierung den Wert pro Zeile verzerrt) und einer Gesamtgröße von 500 erstellt. Der direkte Löser wurde schließlich auf die $(k200 \times k200)$-dimensionalen, rechts oberen Untermatrizen angewandt $(2 \leq k \leq 25)$.

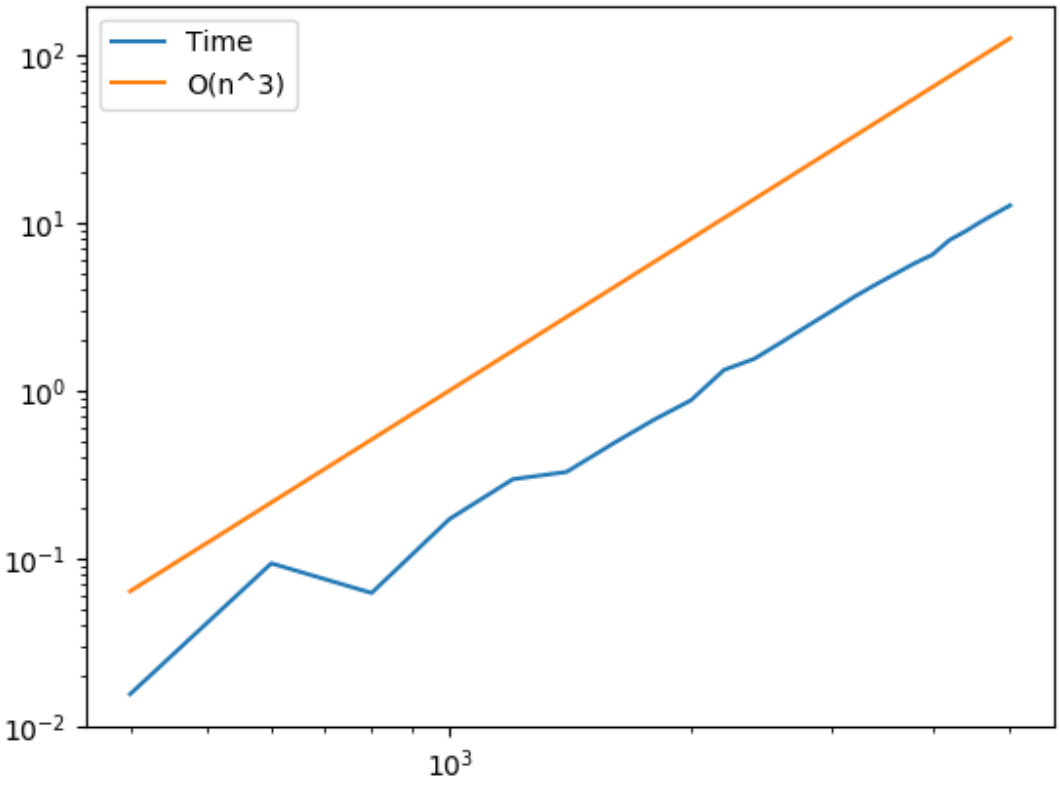


Abbildung 1: Rechenzeit abhängig von der Problemgröße

b)

Implementierung des CG-Verfahrens:

```
def cg(A,b,x0,tol):
    xt = x0
    r0 = b - np.dot(A,xt)
    d = r0
    count = 0
    while(np.linalg.norm(r0) > tol):
        prod = np.dot(np.transpose(r0),r0)
        prod2 = np.dot(A,d)
        alpha = prod/np.dot(np.transpose(d),prod2)
        xt = xt + alpha*d
        r0 = r0 - alpha*prod2
        beta = np.dot(np.transpose(r0),r0)/prod
        d = r0 + beta*d
        count += 1
    print(count)
    return xt
```

Beweis der Äquivalenz obigen Algorithmus zu Algorithmus 8.10 im Numerik-Skript:
Wir führen den Beweis mittels Induktion:
Dabei bezeichnen wir mit * die Variablen aus unserem Algorithmus und ohne * die Variablen des Algorithmus aus dem Skript.
Induktionsanfang:

$$
\begin{aligned}
&A = A^*, b = b^*, x_0 = x_0^* \\
&r_0 = b - Ax_o = b^* - A^* x_0^* = r_0^* \\
&d_0 = r_0 = r_0^* = d_0^* \\
&\alpha_0 = \frac{r_0^T d_0}{d_0^T A d_0} = \frac{r_0^{*T} d_0^*}{d_0^{*T} A d_0^*} = \frac{r_0^{*T} r_0^*}{d_0^{*T} A d_0^*} = \alpha_0^* \\
&x_1 = x_0 + \alpha_0 d_0 = x_0^* + \alpha_0^* d_0^* = x_1^* \\
&r_1 = b - Ax_1 = b^* - A^* x_1^* = b^* - A^* x_0^* - \alpha_0^* A d_0^* = r_0^* - \alpha_0^* d_0^* = r_1^* \\
&\beta_0 = -\frac{r_1^T A d_0}{d_0^T A d_0} = \frac{-r_1^{*T} A d_0^*}{d_0^{*T} A d_0^*} = \frac{-r_1^{*T} A r_0^*}{r_0^{*T} A r_0^*}
\end{aligned}
$$

Unter Ausnutzung der Orthogonalität der Residuen erhalten wir: $r_1^{*T} r_0^* = 0$ und somit können wir den Bruch folgendermaßen erweitern:

$$\frac{-r_1^{*T} A r_0^*}{r_0^{*T} A r_0^*} = \frac{r_1^{*T} r_0^* - \alpha_0^* r_1^{*T} A r_0^*}{\alpha_0^* r_0^{*T} A r_0^*}$$

Nun berechen wir

$$r_1^{*T} r_1^* = r_1^{*T} (r_0^* - \alpha_0^* A d_0^*) = r_1^{*T} r_0^* - \alpha_0^* r_1^{*T} A r_0^*$$

und setzen $\alpha_0^* = \frac{r_0^{*T} r_0^*}{d_0^{*T} A d_0^*}$ ein:

$$
\begin{aligned}
\frac{r_1^{*T} r_0^* - \alpha_0^* r_1^{*T} A r_0^*}{\alpha_0^* r_0^{*T} A r_0^*} = \frac{r_1^{*T} r_1^*}{\frac{r_0^{*T} r_0^*}{r_0^{*T} A r_0^*} r_0^{*T} A r_0^*} = \frac{r_1 * T r_1^*}{r_0^{*T} r_0^*} = \beta_0^* \\
d_1 = r_1 + \beta_0 d_0 = r_1^* + \beta_0^* d_0^* = d_1^*
\end{aligned}
$$

Damit haben wir die Gleichheit der Variablen nach dem ersten Schleifendurchlauf gezeigt.
Sei nun nach n Schleifendurchläufen die Gleichheit aller vorhergehenden Variablen vorausgesetzt:

$$\alpha_{n-1} = \alpha_{n-1}*, x_n = x_n^*, r_n = r_n^*, \beta_{n-1} = \beta_{n-1}*, d_n = d_n^*$$
$$\alpha_n = \frac{r_n^T d_n}{d_n^T A d_n} = \frac{r_n^{*T} d_n^*}{d_n^{*T} A^* d_n^*} = \frac{r_n^{*T}(r_n^* + \beta_{n-1}^* d_{n-1}^*)}{d_n^{*T} A^* d_n^*}$$

Jetzt nutzen wir die Eigenschaft: $\forall 0 \leq j < m : r_m^T d_j = 0$ und erhalten:

$$\frac{r_n^{*T}(r_n^* + \beta_{n-1}^* d_{n-1}^*)}{d_n^{*T} A^* d_n^*} = \frac{r_n^{*T} r_n^*}{d_n^{*T} A^* d_n^*} = \alpha_n^*$$
$$x_{n+1} = x_n + \alpha_n d_n = x_n^* + \alpha_n^* d_n^* = x_{n+1}^*$$
$$r_{n+1} = b - A x_{n+1} = b - A(x_n + \alpha_n d_n) = b - A x_n - \alpha_n d_n =$$
$$= r_n - \alpha_n A d_n = r_n^* - \alpha_n^* A^* d_n^* = r_{n+1}^*$$
$$\beta_n = -\frac{r_{n+1}^T A d_n}{d_n^T A d_n} = \frac{r_{n+1}^{*T} r_n^* - \alpha_n^* r_{n+1}^{*T} A^* d_n^*}{d_n^{*T} A^* d_n^*} = \frac{r_{n+1}*T r_{n+1}^*}{r_n^{*T} r_n^*} = \beta_n^*$$
$$d_{n+1} = r_{n+1} + \beta_n d_n = r_{n+1}^* + \beta_n^* d_n^* = d_{n+1}^*$$

Und der Beweis ist vollständig.

c)

Dünn besetzte Matrizen erlauben effizientere Implementierungen als voll besetzte, indem beim Speichern und Rechnen nur Einträge die ungleich Null sind berücksichtigt werden. Eine Möglichkeit einer solchen Implementierung ist das sogenannte *compressed sparse row* Format. Anstelle aller Einträge $A_{ij}, i, j = 1, dots, n$ einer Matrix $A \in \mathbb{R}^{n \times n}$ werden ein Vektor $v \in \mathbb{R}^{n \times n}$ aller Einträge ungleich Null, ein Vektor $J \in \mathbb{N}_0^m$ von Spaltenindizes und ein Vektor $I \in \mathbb{N}_0^{n+1}$ von Zeigern gespeichert. Die i-te Zeile von A ist dann gegeben durch

$$A_{ij} = \begin{cases} v_{k(j)}, & \text{falls } j \in \{J_{I_i}, J_{I_i} + 1, \ldots, J_{I_i+1} - 1\} \\ 0, & \text{sonst} \end{cases}$$

Implementierung in Python:

```python
class Sparse:

    def __init__(self,b,v, J = np.zeros(0), I = np.zeros(0)):
        if b:
            self.v = np.array(v)
            self.J = np.array(J)
            self.I = np.array(I)
            self.n = len(self.I)-1
        else:
            self.v, self.J, self.I = self.fromdense(v)
            self.n = len(self.I)-1

    def __matmult__(self,b):
        d = np.zeros(self.n)
        for i in range(self.n):
            x = np.array(self.J[self.I[i]:self.I[i+1]]).astype(int)
            d[i] = self.v[self.I[i]:self.I[i+1]]@b[x]
```

```
        return d


    def todense(self):
        A = np.zeros([self.n,self.n])
        for i in range(self.n):
            for j in range(self.I[i],self.I[i+1]):
                A[i][self.J[j]] = self.v[j]
        return A

    def fromdense(self,A):
        v,J = np.zeros(0), np.zeros(0)
        I = np.array([0])
        c = 0
        for i in range(np.shape(A)[0]):
            for j in range((np.shape(A))[0]):
                if A[i][j] != 0:
                    v = np.append(v,A[i][j])
                    J = np.append(J,j)
                    c += 1
            I = np.append(I,c)
        return v, J, I
```

d)

Kombinieren Sie Ihre CG-Implementierung mit Ihrer **sparse** Klasse und testen Sie die Effizienz: Implementierung:

```
def Scg(A,b,x0,tol):
    xt = x0
    r0 = b - A.__matmult__(xt)
    d = r0
    c = 0
    while(np.linalg.norm(r0) > tol):
        prod = np.dot(np.transpose(r0),r0)
        prod2 = A.__matmult__(d)
        alpha = prod/np.dot(np.transpose(d),prod2)
        xt = xt + alpha*d
        r0 = r0 - alpha*prod2
        beta = np.dot(np.transpose(r0),r0)/prod
        d = r0 + beta*d
        c += 1
    print(c)
    return xt
```

Schlechte Nachrichten: Die Effizienz ist gesunken!

e)

Die Konvergenzgeschwindigkeit des CG-Verfahrens ist durch die spektrale Konditionszahl $\text{cond}(A)$ der Matrix A bestimmt. Um die Konvergenzgeschwindigkeit zu erhöhen löst man das vorkonditionierte System

$$D^{-1}AD^{-1T} = D^{-1}b$$

und gewinnt die Lösung x dann durch $x = D^{-1T}y$. Die Matrix D wird dabei so gewählt, dass

- für beliebige Vektoren $z \in \mathbb{R}^n$ der Vektor $D^{-1T}D^{-1}z$ einfach zu berechen ist und
- $\text{cond}(D^{-1}AD^{-1T}) < \text{cond}(A)$.

Implementierung des vorkonditionierten CG-Verfahrens:

```python
def vcg(A,b,x0,P,tol):
    r0 = b - A.__matmult__(x0)
    P_inv = np.linalg.inv(P)
    z0 = P_inv@r0
    d = r0
    c = 0
    while(np.linalg.norm(r0) > tol):
        prod = np.dot(np.transpose(z0),r0)
        prod2 = A.__matmult__(d)
        alpha = np.dot(np.transpose(r0),z0)/np.dot(np.transpose(d),prod2)
        x0 = x0 + alpha*d
        r0 = r0 - alpha*prod2
        z0 = P_inv@r0
        beta = np.dot(np.transpose(z0),r0)/prod
        d = r0 + beta*d
        c += 1
    return xt
```

Wieder schlechte Nachrichten: Der Scheiß kovergiert nicht!

2 Eigenschwingungen

2.1 Aufgabestellung

Das Projekt beschäftigt sich mit den Eigenschwingeungen einer fest eingespannten Saite. Sei dazu $u(t,x)$ die vertikale Auslenkung der Saite an der Position $x \in [0,1]$ zur Zeit t. u wird näherungsweise durch die sogenannte Wellengleichung

$$\frac{\partial^2 u}{\partial x^2}(t,x) = \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}(t,x) \tag{1}$$

für alle $x \in (0,1)$ und $t \in \mathbb{R}$ beschrieben, wobei c die Ausbreitungsgeschwindigkeit der Welle ist. Wenn die Saite an beiden Enden fest eingespannt ist, so gelten die Randbedingungen

$$u(t,0) = u(t,1) = 0$$

für alle $t \in \mathbb{R}$.

Zur Berechnung der Eigenschwingungen suchen wir nach Lösungen u, die in der Zeit harmonisch schwingen. Solche erfüllen folgenden Ansatz

$$u(x,t) = \Re(v(x)e^{-i\omega t})$$

mit einer festen, aber unbekannten Kreisfrequenz $\omega > 0$ und einer Funktion v, welche nur noch vom Ort x abhängt. Durch Einsetzen erhalten wir für v die sogenannte Helmholz-Gleichung

$$-v''(x) = \kappa^2 v(x), \qquad x \in (0,1), \tag{2}$$

mit der unbekannten Wellenzahl $\kappa := \frac{\omega}{c}$ und den Randbedingungen

$$v(0) = v(1) = 0. \tag{3}$$

2.2 Analytische Lösung

$$v_\kappa(x) = C_1 \cos(\kappa x) + C_2 \sin(\kappa x), \qquad x \in [0,1], \tag{4}$$

mit beliebigen Konstanten C_1, C_2 löst die Helmholz-Gleichung (8). Das erkennt man durch stumpfes Einsetzen.

$$\begin{aligned} -v_\kappa''(x) &= -\frac{\partial^2}{\partial x^2}(C_1 \cos(\kappa x) + C_2 \sin(\kappa x)) = -\frac{\partial}{\partial x}(-C_1\kappa \sin(\kappa x) + C_2\kappa \cos(\kappa x)) \\ &= -(-C_1\kappa^2 \cos(\kappa x) - C_2\kappa^2 \sin(\kappa x)) = \kappa(C_1 \cos(\kappa x) + C_2 \sin(\kappa x)) = \kappa^2 v_\kappa(x) \end{aligned}$$

Wir fragen uns, für welche $\kappa > 0$, Konstanten C_1 und C_2 existieren, sodass v_κ auch die Randbedingungen (3) erfüllt.

$$0 \overset{!}{=} \begin{cases} v_\kappa(0) = C_1 \cos 0 + C_2 \sin 0 = C_1 \\ v_\kappa(1) = C_1 \cos\kappa + C_2 \sin\kappa = C_2 \sin\kappa \end{cases}$$

Nachdem $\cos 0 = 1$ und $\sin 0 = 0$, erhält man, aus der oberen Gleichung, $C_1 = 0$. Mit der unteren Gleichung folgt aber auch $C_2 \sin\kappa = 0$. Wenn nun auch $C_2 = 0$, dann erhielte man die triviale Lösung $v_\kappa = 0$. Für eine realistischere Modellierung, d.h. $v_\kappa \neq 0$, müsste $\sin\kappa = 0$, also $\kappa \in \pi\mathbb{Z}$.

Das sind die gesuchten $\kappa > 0$. Sei nun eines dieser κ fest. Offensichtlich ist $C_1 = 0$ eindeutig, $C_2 \in \mathbb{R}$ jedoch beliebig.

2.3 Numerische Approximation

Häufig lassen sich solche Probleme nicht analytisch lösen, sodass auf numerische Verfahren zurückgegriffen wird, welche möglichst gute Näherungen an die exakten Lösungen berechnen sollen. Als einfachstes Mittel dienen sogenannte Differenzenverfahren. Sei dazu $x_j := jh$, $j = 0, \ldots, n$ eine Zerlegung des Intervalls $[0,1]$ mit äquidistanter Schrittweite $h = 1/n$. Die zweite Ableitung in (8) wird approximiert durch den Differenzenquotienten

$$v''(x_j) \approx D_h v(x_j) := \frac{1}{h^2}(v(x_{j-1}) - 2v(x_j) + v(x_{j+1})), \qquad j = 1, \ldots, n-1. \tag{5}$$

Für hinreichend glatte Funktionen v mit einer geeigneten Konstanten $C > 0$ wird der Approximationsfehler quadratisch in h klein, d.h. dass

$$|v''(x_j) - D_h v(x_j)| \leq Ch^2. \tag{6}$$

Es sei zunächst bemerkt, dass (**??**) tatsächlich einen Differenzenquotienten beschreibt. Um das einzusehen, verwenden wir den links- und rechts-seitigen Differenzenquotient erster Ordnung, sowie $x_{j-1} = x_j - h$, $x_{j+1} = x_j + h$. Wir erhalten $\forall\, j = 1, \ldots, n-1$:

$$\begin{aligned} v''(x_j) &= \lim_{h\to 0} \frac{1}{h}(v'(x_j + h) - v'(x_j)) \\ &= \lim_{h\to 0} \frac{1}{h}\left(\frac{1}{h}(v(x_j + h) - v(x_j)) - \frac{1}{h}(v(x_j) - v(x_j - h))\right) \\ &= \lim_{h\to 0} \frac{1}{h^2}(v(x_j + h) - 2v(x_j) + v(x_j - h)) \\ &= \lim_{h\to 0} D_h v(x_j) \end{aligned}$$

Nachdem v hinreichend glatt ist, gilt nach dem Satz von Taylor, dass $\forall\, j = 1, \ldots, n-1$:

$$v(x_j + h) = \sum_{\ell=0}^{n+2} \frac{h^\ell}{\ell!} v^{(\ell)}(x_j) + \mathcal{O}\big(h^{n+3}\big),$$
$$v(x_j - h) = \sum_{\ell=0}^{n+2} \frac{(-h)^\ell}{\ell!} v^{(\ell)}(x_j) + \mathcal{O}\big(h^{n+3}\big).$$

Man beachte, dass sich die ungeraden Summanden, der oberen Taylor-Polynome, sich gegenseitig aufheben. Damit erhalten wir für den Differenzenquotient $D_h v(x_j)$, $j = 1, \ldots, n-1$ eine asymptotische Entwicklung.

$$\begin{aligned} D_h v(x_j) &= \frac{1}{h^2}(v(x_j - h) + v(x_j + h) - 2v(x_j)) \\ &= \frac{1}{h^2}\left(2v(x_j) + h^2 v''(x_j) + \sum_{\substack{\ell=4 \\ \ell \in 2\mathbb{N}}}^{n+2} \frac{h^\ell}{\ell!} v^{(\ell)}(x_j)(1 + (-1)^\ell) - 2v(x_j)\right) + \mathcal{O}\big(h^{n+3}\big) \\ &= v''(x_j) + 2\sum_{\ell=1}^{\lfloor \frac{n}{2} \rfloor} \frac{h^{2\ell}}{(2\ell+2)!} v^{(2\ell)}(x_j) + \mathcal{O}\big(h^{n+1}\big) \end{aligned}$$

Daraus folgt unmittelbar die quadratische Konvergenz (6), $\forall\, j = 1, \ldots, n-1$:

$$D_h v(x_j) - v''(x_j) = \mathcal{O}\big(h^2\big), \qquad h \to 0.$$

Wir wollen nun den Differenzenquotienten $D_h v(x_j)$ verwenden, um ein Eigenwertproblem der Form $A\vec{v} = \lambda\vec{v}$ mit einer Matrix $A \in \mathbb{R}^{(n-1)\times(n-1)}$ zu dem Eigenvektor $\vec{v} := (v(x_1, \ldots, v(x_{n-1}))^T)$ und dem Eigenwert $\lambda := -\kappa^2$ herzuleiten.

Es wird eine Matrix A_n gesucht, die den Differenzenquotienten $D_h v(x_j)$ auf den Vektor $\vec{v}$ komponentenweise anwendet. Wir rufen in Erinnerung, dass $h = 1/n$ und definieren die naheliegende Matrix

$$A_n := \frac{1}{h^2}\begin{pmatrix} -2 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{pmatrix} \in \mathbb{R}^{(n-1)\times(n-1)}.$$

Weil nun die Randbedingungen (3) gelten, d.h. $v(x_0), v(x_n) = 0$, leistet diese Matrix A_n tatsächlich das Gewünschte.

$$A_n\vec{v} = \frac{1}{h^2}\begin{pmatrix} v(x_0) - 2v(x_1) + v(x_2) \\ v(x_1) - 2v(x_2) + v(x_3) \\ \vdots \\ v(x_{n-3}) - 2v(x_{n-2}) + v(x_{n-1}) \\ v(x_{n-2}) - 2v(x_{n-1}) + v(x_{n-0}) \end{pmatrix} = \begin{pmatrix} D_h v(x_1) \\ \vdots \\ D_h v(x_{n-1}) \end{pmatrix}$$

Das Eigenwertproblem wurde mit `np.linalg.eig`, für beliebige $n \geq 2$, gelöst. Wir vergleichen die Eigenwerte und Eigenvektoren mit den analytischen Ergebnissen.

Betrachtet man die, unten aufgelisteten, Eigenwerte, der ersten paar Matrizen $A_2, \dots, A_{10}$, so legen diese ein gewisses (quadratisches) Konvergenzverhalten nahe. Die Matrix A_n besitzt also scheinbar $n-1$ paarweise verschiedene Eigenwerte $\lambda_{1,n} > \cdots > \lambda_{n-1,n}$, welche jeweils gegen $\lambda_j := -(\pi j)^2$, $j \in \mathbb{N}$ konvergieren.

$$\lambda_{j,n} \xrightarrow{n\to\infty} \lambda_j$$

. . . .

Wir bezeichnen mit $\epsilon_j(n) := |\lambda_j - \lambda_{j,n}|$, $j = 1, \dots, n-1$ den absoluten Konvergenz-Fehler des j-ten Eigenwertes. In der folgenden Abbildung **??** wurde dieser für $j = 1, 2, 3$ gegen id^2, doppelt logarithmisch, geplottet.

Allem Anschein nach, verschwindet ϵ_j quadratisch. Das korreliert mit dem Ergebnis (6). Man beachte, dass der j-te Eigenwert erst ab einer Matrix A_n, $n > j$ existiert. Daher fangen die plots von ϵ_j desto später an, je größer j ist. Für größeres j ist auch der intitiale Fehler größer. Obwohl dieser ebenfalls quadratisch konvergiert, werden mehr Rechenoperationen für ein genaues Ergebnis benötigt.

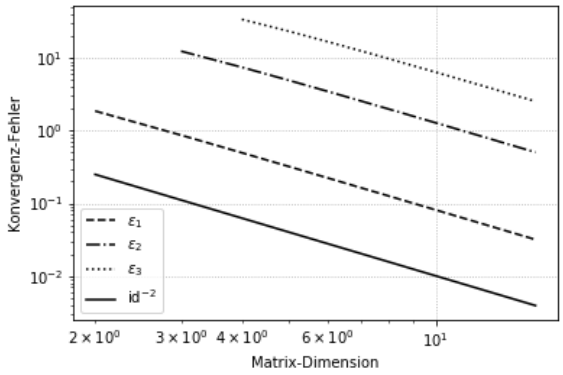


Abbildung 2: Konvergenz-Fehler der Eigenwerte von A_n

Wenn wir mit dem plot von den Eigenvektoren fertig sind, dann kommt der auch noch hier her!

2.4 verallgemeinerte Eigenschwingungen

Die Ausbreitungsgeschwindigkeit c in (1) hängt vom Material der Saite ab. Bisher haben wir sie als konstant angenommen, d.h. die Saite bestand aus einem Material. Sei nun für $c_0, c_1 \in \mathbb{R}$

$$c(x) := \begin{cases} c_0, x \in (0, 1/2) \\ c_1, x \in (1/2, 1) \end{cases} . \tag{7}$$

Zuerst leiten wir eine zur Helmholz-Gleichung (8) ähnliche Gleichung her und geben einen (4) entsprechenden Lösungsansatz an, wenn die Lösung v auf $(0, 1)$ stetig differenzierbar sein soll. Dabei betrachten wir eine angepasste Version der Wellengleichung (1).

$$\frac{\partial^2 u}{\partial x^2}(t,x) = \frac{1}{c^2(x)}\frac{\partial^2 u}{\partial t^2}(t,x), \qquad x \in (0,1), \qquad t \in \mathbb{R}$$

Wir verwenden jedoch den selben Ansatz, wie Vorher. Das war $u(x,t) = \Re(v(x)e^{-i\omega t})$, mit einer festen, aber unbekannten Kreisfrequenz $\omega > 0$ und einer Funktion v, welche nur noch vom Ort x abhängt.

Einsetzen und analoges Nachrechnen, gibt mit der unbekannten Wellenzahl $\kappa(x) := \frac{\omega}{c(x)}$, die Randbedingungen (3) und

$$-v''(x) = \kappa^2(x)v(x), \qquad x \in (0,1). \tag{8}$$

Um Probleme mit der Differenzierbarkeit von κ zu vermeiden, führen wir die Abkürzungen $\kappa_0 := \frac{\omega}{c_0}$, $\kappa_1 := \frac{\omega}{c_1}$ ein. Wir definieren den Lösungsansatz durch Fallunterscheidung und mit (vorerst) beliebigen Konstanten $C_{01}, C_{02}, C_{11}, C_{12}$.

$$v(x) := \begin{cases} C_{01}\cos(\kappa_0 x) + C_{02}\sin(\kappa_0 x), & x \in (0,1/2) \\ C_{11}\cos(\kappa_1 x) + C_{12}\sin(\kappa_1 x), & x \in (1/2,1) \end{cases}$$

Durch Berücksichtigung der Randbedingungen (3), erhält man (fast analog zu Vorher)

$$C_{01} = 0, \qquad C_{11}\cos\kappa_1 + C_{12}\sin\kappa_1 = 0.$$

Soll v auf $1/2$ stetig fortgesetzt werden, so müssen dessen links- und rechts-seitiger Grenzwert übereinstimmen.

$$C_{02}\sin(\kappa_0/2) = \lim_{x\to 1/2-} v(x) = \lim_{x\to 1/2+} v(x) = C_{11}\cos(\kappa_1/2) + C_{12}\sin(\kappa_1/2)$$

Um stetige Differenzierbarkeit zu erhalten, muss auch die Ableitung

$$v'(x) = \begin{cases} C_{02}\kappa_0\cos(\kappa_0 x), & x \in (0,1/2) \\ -C_{11}\kappa_1\sin(\kappa_1 x) + C_{12}\kappa_1\cos(\kappa_1 x), & x \in (1/2,1) \end{cases}$$

auf $1/2$ stetig fortgesetzt werden.

$$\kappa_0 C_{02}\cos(\kappa_0/2) = \lim_{x\to 1/2-} v'(x) = \lim_{x\to 1/2+} v'(x) = -C_{11}\kappa_1\sin(\kappa_1/2) + C_{12}\kappa_1\cos(\kappa_1/2)$$

Aus den Randbedingungen und stetigen Fortsetzungen, ergibt sich also das homogene lineare Gleichungssystem $R\vec{C} = 0$, mit

$$R := \begin{pmatrix} \sin(\kappa_0/2) & -\cos(\kappa_1/2) & -\sin(\kappa_1/2) \\ \kappa_0\cos(\kappa_0/2) & \kappa_1\sin(\kappa_1/2) & -\kappa_1\cos(\kappa_1/2) \\ 0 & \cos\kappa_1 & \sin\kappa_1 \end{pmatrix} \in \mathbb{R}^{3\times 3}, \qquad \vec{C} := \begin{pmatrix} C_{02} \\ C_{11} \\ C_{12} \end{pmatrix} \in \mathbb{R}^{3\times 1}.$$

Sei $R \in \mathrm{GL}_3(\mathbb{R})$ regulär, so ist deren Kern trivial, d.h. $\ker R = \{0\}$, und somit auch die Lösung $\vec{C} = 0$. Dieser Fall wäre jedoch wieder der trivialfall, darum betrachten wir den anderen Fall, also dass $\det R = 0$ gilt. Mit SymPy berechnet man

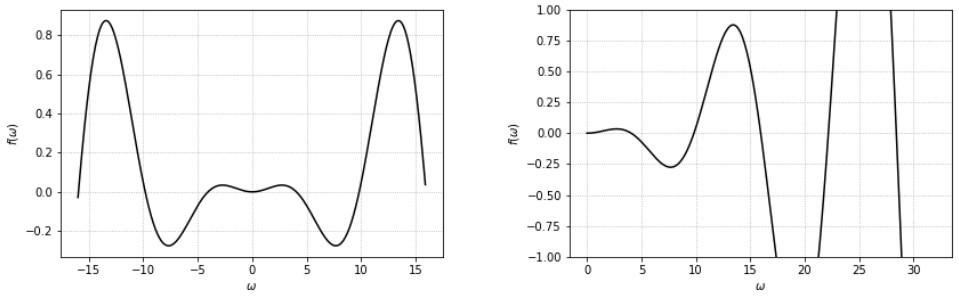
$$\det R = \sin\left(\frac{\kappa_0}{2}\right)\cos\left(\frac{\kappa_1}{2}\right)\kappa_1 + \sin\left(\frac{\kappa_1}{2}\right)\cos\left(\frac{\kappa_0}{2}\right)\kappa_0.$$

Über $\kappa_0 = \frac{\omega}{c_0}$, $\kappa_1 = \frac{\omega}{c_1}$ lässt sich ω, für gegebene c_0, c_1 als (nicht eindeutige) Nullstelle einer Funktion f charakterisieren. Diese Überlegung, wird, in Form von folgender Funktion, implementiert.

```python
# c ... pair of propagation speeds

def get_zero_function(c):

    # allocate some sympy symbols:
    omega = sp.Symbol('\omega')
    kappa = sp.IndexedBase('\kappa')

    # implement the matrix R (properly):
    R = sp.Matrix([[ sp.sin(kappa[0]/2),  kappa[0]*sp.cos(kappa[0]/2), 0],
                   [-sp.cos(kappa[1]/2),  kappa[1]*sp.sin(kappa[1]/2), sp.cos(kappa[1])],
                   [-sp.sin(kappa[1]/2), -kappa[1]*sp.cos(kappa[1]/2), sp.sin(kappa[1])]])
    R = R.T

    # calculate R's determinant (properly):
    det = sp.det(R)
    det = sp.simplify(det)

    # substitute for kappa_0 and kappa_1:
    kappa_0 = omega/c[0]
    kappa_1 = omega/c[1]
    substitution = {kappa[0]: kappa_0, kappa[1]: kappa_1}
    det = det.subs(substitution)

    # transform expression det into proper numpy function:
    zero_function = sp.lambdify(omega, det, 'numpy')

    return zero_function
```

Nun können wir f für fixe c_0, c_1 plotten lassen. Dann bekommen wir ein besseres Verständnis dafür, welchen Startwert wir wählen sollen, um die Gleichung $f(\omega) = 0$, mit `fsolve` aus `scipy.optimize`, lösen zu lassen.

Für die folgenden Plots in Abbildung 3, wurden die arbiträren Werte $c_0 = 100$, $c_1 = 1$ gewählt. Die Funktion f ist scheinbar gerade, d.h. symmetrisch bzgl. der y-Achse.



(a) auf dem Intervall $(-16, 16)$ (b) auf dem Intervall $(0, 64)$ und herangezoomt

Abbildung 3: Plots von f für $c_0 = 100$, $c_1 = 1$

Dementsprechend, können passende Startwerte für iterative Verfahren gewählt werden. Die Ergebnisse vom,

bereits erwähnten, `fsolve` folgen. Warum die Quadrate dieser Ergebnisse eine Rolle spielen, wird später noch erwähnt.

Wir wollen nun den Differenzenquotienten $D_h v(x_j)$, um ein verallgemeintertes Eigenwertproblem der Form $A\vec{v} = \lambda B\vec{v}$ mit Matrizen $A, B \in \mathbb{R}^{(n-1)\times(n-1)}$ herzuleiten.

Sei abermals $x_j := jh$, $j = 0, \dots, n$ unsere Zerlegung des Intervalls $[0, 1]$ mit äquidistanter Schrittweite $h = 1/n$. Die Matrix A_n, für den Differenzenquotienten $D_h v(x_j)$ und der Vektor $\vec{v} := (v(x_1), \dots, v(x_{n-1}))^T$, bleiben ebenfalls nach wie vor so, wie sie waren.

$B_n\lambda$ soll nun, analog zu Vorher, $-\kappa^2$ repräsentieren. Diesmal jedoch, ist κ als Funktion zu verstehen. Also wird die Matrix B_n deren Fallunterscheidungen übernehmen und λ bleibt konstant.

$$B_n := \operatorname{diag}^{-2}(\underbrace{c_0, \dots, c_0}_{\lfloor \frac{n-1}{2} \rfloor\text{-mal}}, \underbrace{c_1, \dots, c_1}_{\lceil \frac{n-1}{2} \rceil\text{-mal}}), \qquad \lambda := -\omega^2.$$

Die Wahl von B_n lässt sich wie folgt begründen: Für zwei Vektoren a, b, ist die Matrix-Vektor-Multiplikation $\cdot$, mit der Diagonalmatrix von a, äquivalent zur komponentenweisen Multiplikaiton $\odot$, d.h. $\operatorname{diag}(a) \cdot b = a \odot b$. Bei dem vorherigen Eigenwert-Problem wäre B_n als die Einheits-Matrix I_n zu betrachten, nachdem der Eigenwert λ gleich ganz $-\kappa^2$ (konstant) übernehmen konnte. Da $-\kappa^2$ nun zwei Werte

$$-\left(\frac{\omega}{c_0}\right)^2, -\left(\frac{\omega}{c_1}\right)^2$$

annimmt, muss diese Eigenschaft von Diagonalmatrizen ausgenutzt werden, um die Unterscheidung zwischen $x_j < 1/2$ und $x_j \geq 1/2$ zu realisieren.

Wenn wir davon ausgehen, dass $c_0, c_1 \neq 0$, so ist B_n wohldefiniert als Diagonalmatrix deren quadratische Kehrwerte. Damit lässt sich dieses verallgemeinerte Eigenwertproblem $A_n\vec{v} = \lambda B_n\vec{v}$, mit nahezu keinem Aufwand, in ein konkretes $B_n^{-1}A_n\vec{v} = \lambda\vec{v}$ umformulieren.

Dieses Eigenwertproblem wurde ebenfalls mit `np.linalg.eig`, für beliebige $n \geq 2$, gelöst. Wir vergleichen die Eigenwerte und Eigenvektoren mit den oberen „analytischen“ Ergebnissen mit $c_0 = 100$, $c_1 = 1$. Dass die Eigenwerte gegen $-\omega^2$ konvergieren, überrascht wenig.

```
n = 1000
-----
Eigenvalues:
-16.44054324040394
-96.3725475394107
-254.1214683324469
```

3 Titel