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)

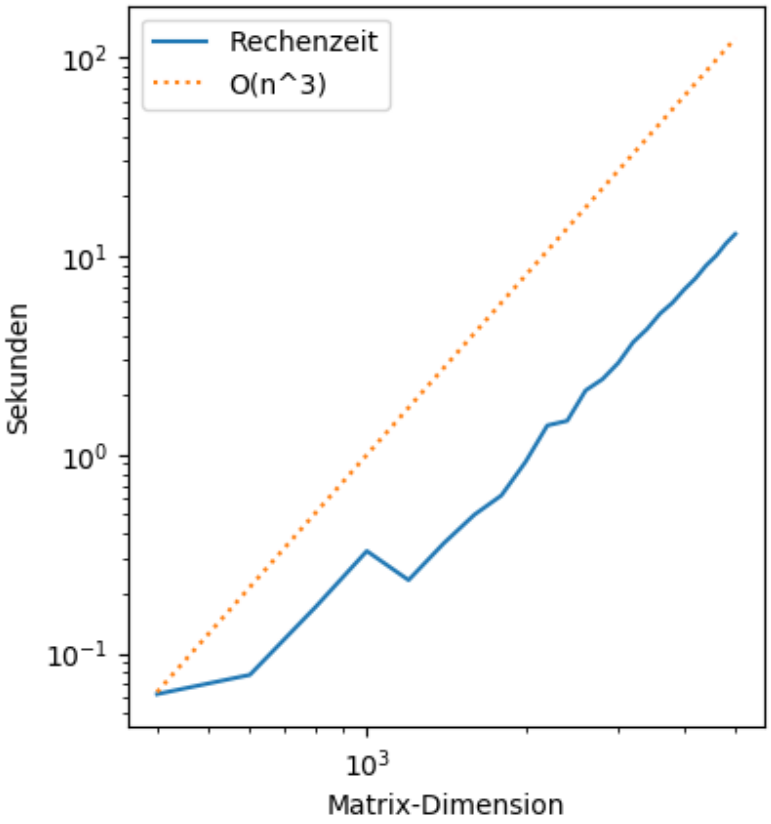
Generieren Sie eine symmetrisch positiv definite Zufallsmatrix $A \in \mathbb{R}^{n \times n}$, wobei pro Zeile eine fixe Anzahl an Einträgen ungleich Null sind. Testen Sie für eine zufällige rechte Seite $b \in \mathbb{R}^n$ bis zu welcher Größe n das lineare Gleichungssystem

$$Ax = b$$

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 5000 erstellt. Der direkte Löser wurde schließlich auf die $(200k \times 200k)$-dimensionalen Ausschnitte der oberen rechten Ecke angewandt $(2 \leq k \leq 25)$. Wie man sieht erreichen wir damit schon langsam die Grenze des akzeptabel Berechenbaren, für die volle 5000×5000-Matrix braucht der Algorithmus schon über 10 Sekunden.



	Zeit
400	0.0625
600	0.078125
800	0.171875
1000	0.328125
1200	0.234375
1400	0.359375
1600	0.5
1800	0.625
2000	0.921875
2200	1.40625
2400	1.484375
2600	2.109375
2800	2.40625
3000	2.90625
3200	3.6875
3400	4.296875
3600	5.140625
3800	5.828125
4000	6.796875
4200	7.71875
4400	8.96875
4600	10.0625
4800	11.59375
5000	12.9375

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
    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
    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:

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

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:

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

Und der Beweis ist vollständig.

Rechenschritte pro Schleifendurchlauf:
Am aufwändigsten ist natürlich die Matrix-Vektor-Multiplikation, wovon wir in unserem Algorithmus im Gegensatz zu jenem aus dem Skript nur eine pro Schleifendurchlauf benötigen. Zusätzlich dazu brauchen wir 3 Vektor-Vektor-Multiplikationen und 4 Vektor-Skalar-Operationen.
Damit erhalten wir insgesamt $n^2 + 7n$ Flops pro Durchlauf.
Im Vergleich dazu verwendet der Algorithmus aus dem Numerik-Skript pro Iteration zwei Matrix-Vektor-Multiplikationen und ist daher aus Effizienzgründen unterlegen.

Die Theorie besagt, dass das CG-Verfahren spätentens nach n Durchläufen die exakte Lösung liefert und für die Iterierten folgende Fehlerabschätzung gilt:

$$\left\| x^{(t)} - A^{-1}b \right\|_A \leq 2 \left(\frac{1 - 1/\sqrt{\kappa}}{1 + 1/\sqrt{\kappa}} \right)^t \left\| x^{(0)} - A^{-1}b \right\|_A, \quad t \in \mathbb{N},$$

mit der spektralen Konditionszahl $\kappa = \text{cond}_2(A)$.
Also sollte das Verfahren exponentiell kovergieren ($\mathcal{O}(AB^t)$) mit Konstanten
$A = 2\|x^{(0)} - A^{-1}b\|_A, \quad B = \frac{1-1/\sqrt{\kappa}}{1+1/\sqrt{\kappa}}$
Wie in untenstehender Grafik ersichtlich scheint bei unseren Zufallsmatrizen der Wert von B sich in etwa bei 0.92 einzupendeln und unsere vorgegebene Toleranz von 10^{-8} wird bereits nach etwa 350 Iterationen erreicht, noch weit vor dem theoretisch (bis auf Rechenfehler) garantierten exakten Resultat nach $n = 5000$ Durchläufen. Ansonsten verhält sich der Fehler wie wir ihn erwarten würden.
Unsere Testwerte:

```
n = 5000
A = np.random.rand(n,n)
A = np.dot(A,np.transpose(A))
A += 10*np.diag(abs(np.random.random(n)))
b = np.random.rand(n)
tol = 10**(-8)
x0 = np.random.rand(n)*10
```

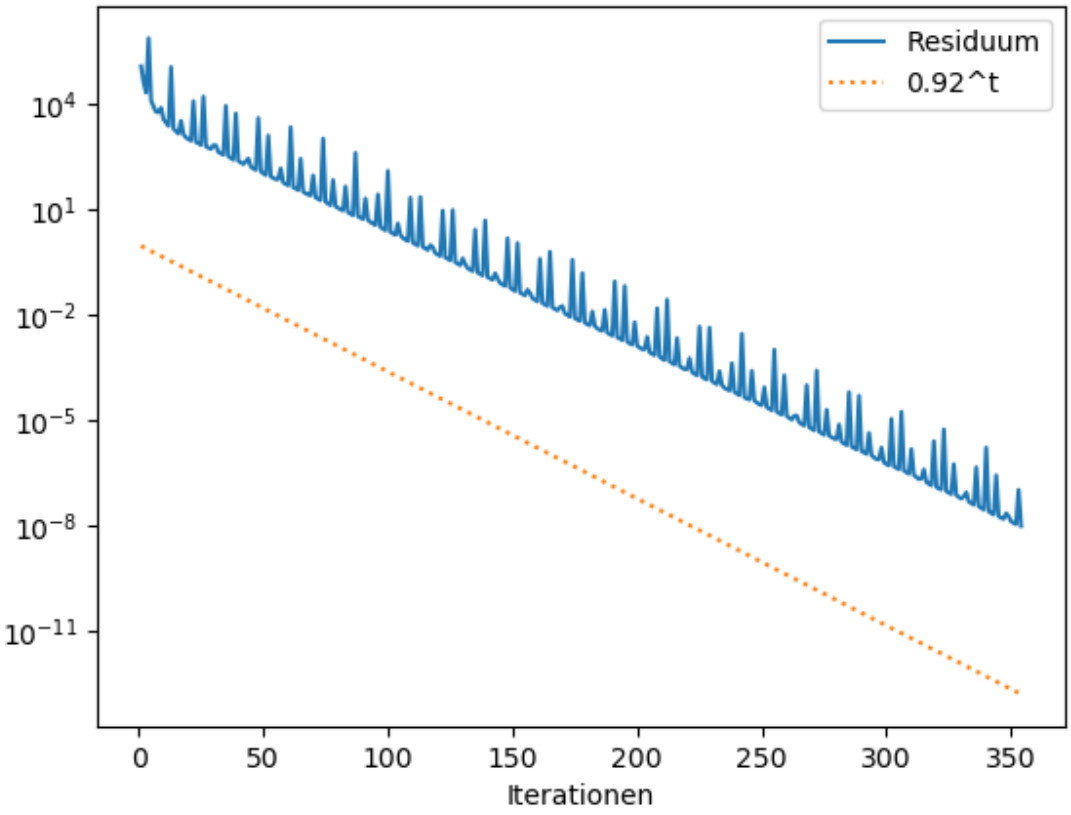


Abbildung 2: Residuum abhängig von der Anzahl an Iterationen

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 $A_{ij}, i, j = 1, \ldots, n$ 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
```

Da wir bei diesem Projekt uns nur auf das cg-Verfahren konzentrieren, haben wir nur die Matrix-Vektor Multiplikation für die Klasse Sparse implementiert.

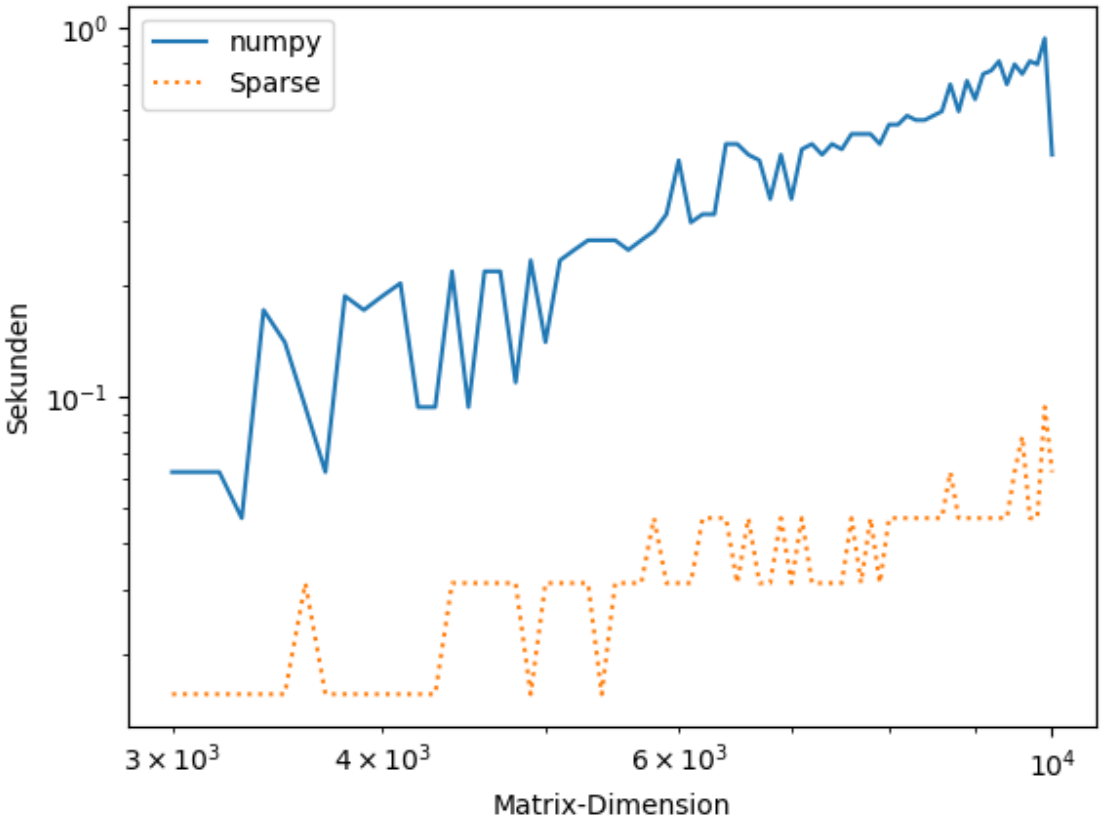


Abbildung 3: Vergleich Numpy Matrixmultiplikation vs. Sparse Matrixmultiplikation

Hier sehen wir, dass erst ab einer gewissen Größe die Sparse-Multiplikation effizienter gegenüber der Numpy-Multiplikation ist.

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
    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
    return xt
```

Bei dieser CG-Implementierung verwenden wir anstelle der Numpy-Matrix-Vektor-Multiplikation die Implementierung von der Sparse-Klasse. Zu beachten ist, dass die Matrix A ein Objekt der Klasse Sparse sein muss, damit die Funktion durchgeführt werden kann. Ansonsten ist die Implementierung ident zum vorherigen cg-Verfahren.

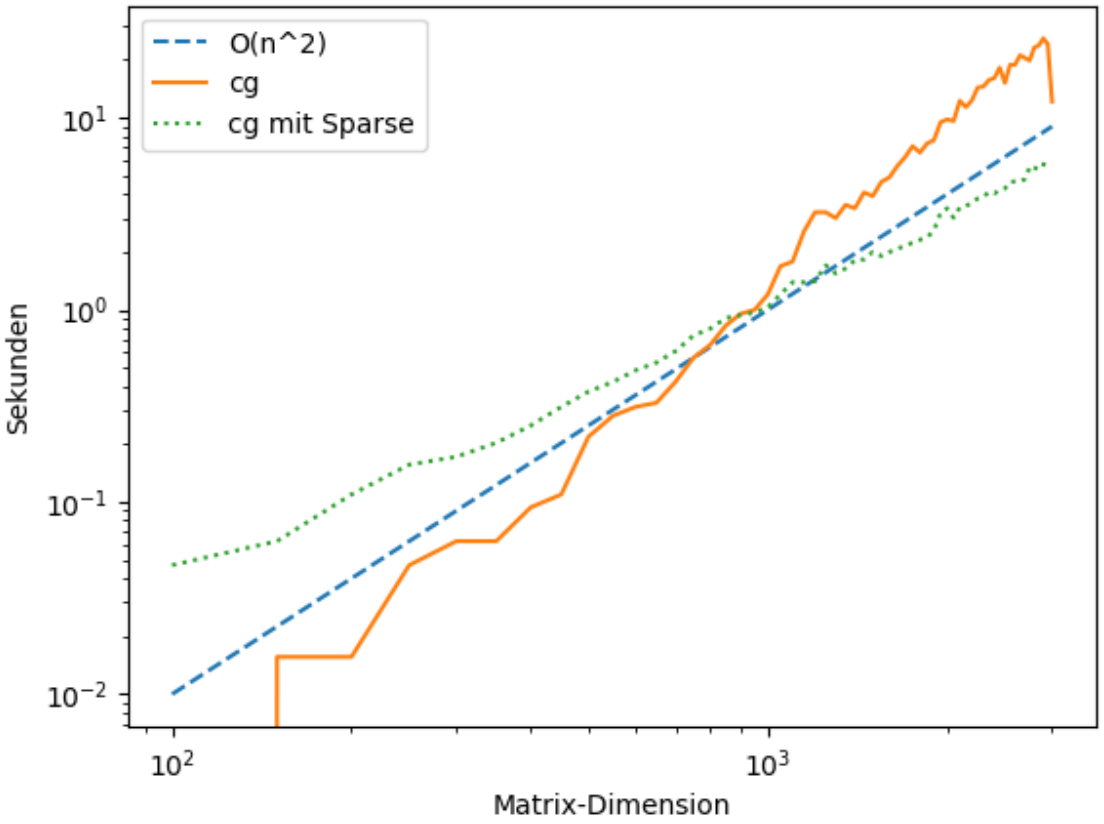


Abbildung 4: cg-Verfahren vs. cg-Verfahren mit Klasse Sparse

Ab einer Matrixgröße von ca. 1000 ist das Scg-Verfahren effizienter als die übliche Implementierung. Man sieht außerdem, dass beide Algorithmen eine Laufzeit von etwa $\mathcal{O}(n^2)$

e)

Die Konvergenzgeschwindigkeit des CG-Verfahrens ist durch die spektrale Konditionszahl 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:

```
    def vcg(A,b,x0,P,tol):
        r0 = b - A.__matmult__(x0)
        P_inv = np.linalg.inv(P)
        z0 = P_inv@r0
        d = z0
        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 = z0 + beta*d
    return x0
```

Wieder schlechte Nachrichten: Der Scheiß kovergiert!

2 Eigenschwingungen

2.1 Problembeschreibung

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

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

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 (2). 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 (2) wird approximiert durch den Differenzenquotienten

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

Es sei angemerkt, dass (5) 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 $\forall\, j = 1, \ldots, n-1$:

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

2.3.1 Approximationsfehler

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

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

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

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

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

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

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

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

2.3.2 Eigenwertproblem

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

$\lambda := \kappa^2$ herzuleiten. Das soll die Helmholz-Gleichung (2), mit Tupeln und Eigenwerten für die Funktionen v_κ bzw. Vorfaktoren κ, approximieren.

Es wird eine Matrix $-A_n$, $n \geq 2$ gesucht, die den Differenzenquotienten $D_h v(x_j)$ auf den oberen Vektor $\mathbf{v}^{(n)}$ komponentenweise anwendet. Wir rufen in Erinnerung, dass $h = 1/n$ und definieren die naheliegende Matrix

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

Wenn nun die Randbedingungen (3) gelten sollen, d.h. $v(x_0), v(x_n) = 0$, leistet diese Matrix A_n tatsächlich das Gewünschte. Sie approximiert die linke Seite der Helmholz-Gleichung (2).

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

Die Matrix A_n wurde in der Funktion `my_numpy_matrix` implementiert.

```
def my_numpy_matrix(n):

    assert n >= 2

    h = 1/n

    a = -2 * np.ones(n-1)
    b = np.ones(n-2)

    A = np.diag(b, -1) + np.diag(a, 0) + np.diag(b, 1)
    A = -A/h**2

    return A
```

Das Eigenwertproblem wurde mit `np.linalg.eig`, für beliebige $n \geq 2$, gelöst. `np.linalg.eig` retourniert die Eigenwerte und Eigenvektoren nicht zwangsläufig, der größe der Eigenwerte nach, sortiert. Nachdem der Zusammenhang zwischen Eigenwert und Eigenvektor nicht verloren gehen soll, erfordert dies einigen logistischen Aufwand. Das ist aber nicht wesentlich und wird daher nicht weiter erklärt. Wir vergleichen lieber die Eigenwerte und Eigenvektoren mit den analytischen Ergebnissen.

In der folgenden Tabelle, sind die 3 Eigenwerte (sollten diese bereits existieren), der Matrizen $A_2, \ldots, A_{10}$, aufgelistet. Diese Tabelle ist analog zu (7) zu verstehen.

	2	3	4	5	6	7	8	9	10
1	8.0	9.0	9.372583	9.54915	9.646171	9.705051	9.743420	9.769795	9.788697
2	NaN	27.0	32.000000	34.54915	36.000000	36.897999	37.490332	37.900800	38.196601
3	NaN	NaN	54.627417	65.45085	72.000000	76.192948	79.016521	81.000000	82.442950

Die Matrix A_n besitzt also scheinbar $n-1$ paarweise verschiedene Eigenwerte $\lambda_{1,n} < \cdots < \lambda_{n-1,n}$, welche jeweils gegen $\lambda_i := (i\pi)^2$, $i \in \mathbb{N}$ konvergieren.

$$\begin{array}{ccccccccc} \lambda_{1,2} & \to & \lambda_{1,3} & \to & \cdots & \to & \lambda_{1,n} & \xrightarrow{n\to\infty} & \lambda_1 = (1\pi)^2 \\ & & \lambda_{2,3} & \to & \cdots & \to & \lambda_{2,n} & \xrightarrow{n\to\infty} & \lambda_2 = (2\pi)^2 \\ & & & & \ddots & \ddots & \vdots & \vdots & \vdots \\ & & & & & & \lambda_{i,n} & \xrightarrow[n\to\infty]{} & \lambda_i = (i\pi)^2 \end{array} \tag{7}$$

Nachdem $\left\{\sqrt{\lambda_i} : i \in \mathbb{N}_0\right\} = \pi\mathbb{N}_0 \subsetneq \pi\mathbb{Z}$, könnte man sich nun fragen, wo die andere Hälfte der analytischen Ergebnisse steckt. Die Erklärung ist ganz unspektakulär $\lambda := (\pm\kappa)^2$.

Wir bezeichnen mit $\epsilon_i(n) := |\lambda_i - \lambda_{i,n}|$, $i = 1, \ldots, n-1$ den absoluten Konvergenz-Fehler des i-ten Eigenwertes. In der folgenden Abbildung 5 wurde ϵ_i mit der Vergleich-Geraden id^2, doppelt logarithmisch, geplottet.

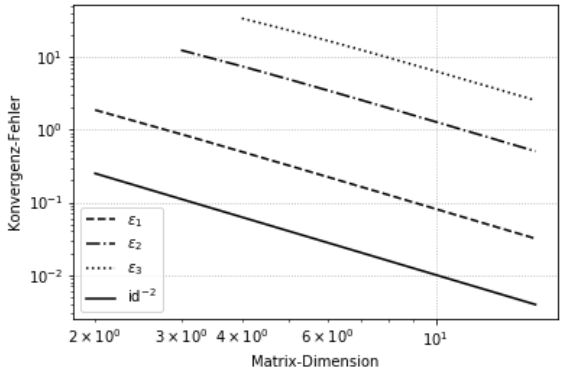
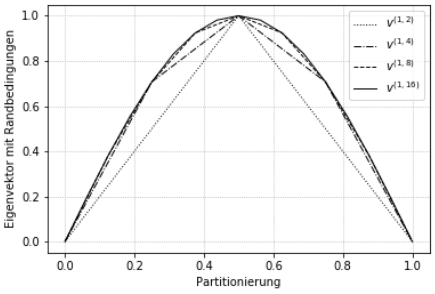


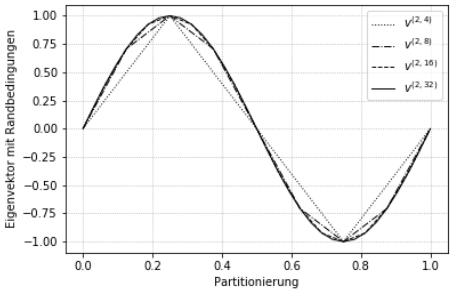
Abbildung 5: Konvergenz-Fehler der Eigenwerte von A_n

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

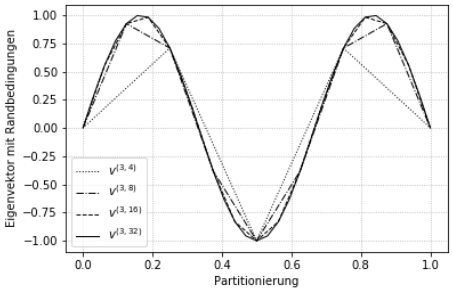
Seien $\mathbf{v}^{(1,n)}, \ldots, \mathbf{v}^{(n-1,n)}$ die Eigenvektoren (modulo Konstante), zu den Eigenwerten $\lambda_{1,n} < \cdots < \lambda_{n-1,n}$, der Matrix A_n. Diese sollten nun gegen die Funktionen v_{κ_i}, $\kappa_i = \sqrt{\lambda_i}$, vielleicht sogar quadratisch, konvergieren. Folgende Abbildungen sollen dies veranschaulichen.



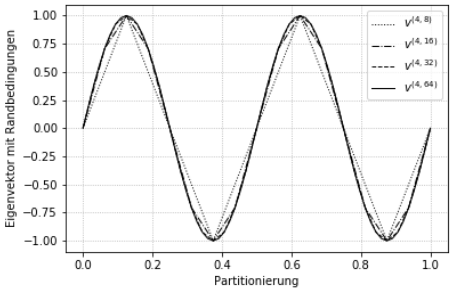
(a) Eigenvektoren $\mathbf{v}^{(1,n)}$, $n = 2, 4, 8, 32$



(b) Eigenvektoren $\mathbf{v}^{(2,n)}$, $n = 4, 8, 16, 64$



(c) Eigenvektoren $\mathbf{v}^{(3,n)}$, $n = 8, 16, 32, 64$



(d) Eigenvektoren $\mathbf{v}^{(4,n)}$, $n = 8, 16, 32, 64$

Abbildung 6: Eigenvektoren $\mathbf{v}^{(i,n)}$, $i = 1, \ldots, 4$ der Matrizen A_n

Erstaunlicherweise, gibt es scheinbar keinen Konvergenz-Fehler, da die Eigenvektoren direkt an den Grenzfunktionen liegen. Mit anderen Worten, $\forall\, n \in \mathbb{N}, \forall\, i, j = 1, \ldots n-1$:

$$\mathbf{v}_j^{(i,n)} = v_{\kappa_i}(x_j).$$

Anschaulich, erhält man ein zu (7) analoges Schema (8).

$$\begin{array}{ccccccccc} \mathbf{v}^{(1,2)} & \to & \mathbf{v}^{(1,3)} & \to & \cdots & \to & \mathbf{v}^{(1,n)} & \xrightarrow[n\to\infty]{} & v_{\kappa_1} \\ & & \mathbf{v}^{(2,3)} & \to & \cdots & \to & \mathbf{v}^{(2,n)} & \xrightarrow[n\to\infty]{} & v_{\kappa_2} \\ & & & & \ddots & \ddots & \vdots & \vdots & \vdots \\ & & & & & & \mathbf{v}^{(i,n)} & \xrightarrow[n\to\infty]{} & v_{\kappa_i} \end{array} \tag{8}$$

2.4 Verallgemeinerte Problembeschreibung

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 $c_0, c_1 \in \mathbb{R}$

$$c(x) := \begin{cases} c_0, x \in (0, 1/2) \\ c_1, x \in (1/2, 1) \end{cases} . \tag{9}$$

2.5 Verallgemeinerte Analytische Lösung

Zuerst leiten wir eine zur Helmholz-Gleichung (2) 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

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

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, und 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

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

auf $1/2$ stetig fortgesetzt werden.

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

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

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

Sei $R \in \mathrm{GL}_3(\mathbb{R})$ regulär, so ist deren Kern trivial, d.h. $\ker R = \{0\}$, und somit auch die Lösung $\mathbf{C} = 0$. Dieser Trivialfall wurde jedoch vorhin bereits ausgeschlossen. Darum betrachten wir $\det R = 0$. Mit SymPy berechnet man

$$\begin{aligned}\det R &= \sin\left(\frac{\kappa_0}{2}\right)\cos\left(\frac{\kappa_1}{2}\right)\kappa_1 + \sin\left(\frac{\kappa_1}{2}\right)\cos\left(\frac{\kappa_0}{2}\right)\kappa_0 \\ &= \sin\left(\frac{\omega}{2c_0}\right)\cos\left(\frac{\omega}{2c_1}\right)\frac{\omega}{c_1} + \sin\left(\frac{\omega}{2c_1}\right)\cos\left(\frac{\omega}{2c_0}\right)\frac{\omega}{c_0} =: f_c(\omega) \end{aligned} \tag{10}$$

Für feste c_0, c_1, lässt sich das gewünschte ω, als (nicht eindeutige) Nullstelle dieser Funktion f_c, charakterisieren.

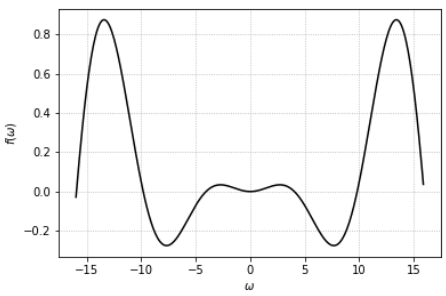
2.6 Verallgemeinerte Semi-Analytische Lösung

Das Ergebnis aus (10) wurde, in Form von folgender Funktion, implementiert.

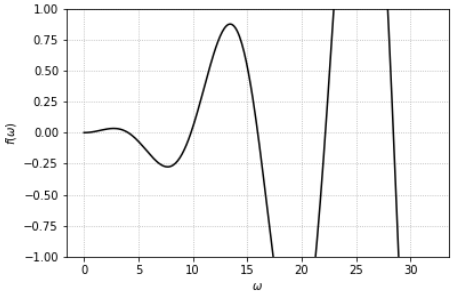
```
# c ... pair of propagation speeds

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

Für die folgenden Plots in Abbildung 7, wurden die arbiträren Werte $c_0 = 100$, $c_1 = 1$ gewählt. Die davon abhängige Funktion f_c ist scheinbar gerade. Das liegt an (10), sowie dass cos gerade und sin ungerade ist.



(a) auf dem Intervall $(-16, 16)$



(b) auf dem Intervall $(0, 30)$ und herangezoomt

Abbildung 7: Plots von f_c für $c_0 = 100$, $c_1 = 1$

Dementsprechend, können passende Startwerte $\tilde{\omega}$ für iterative Verfahren gewählt werden. Die jeweils ersten Ergebnisse ω vom, bereits erwähnten, `scipy.optimize.fsolve` sind in der folgenden Tabelle. Die Quadrate ω^2 dieser Ergebnisse sind Approximationen der Grenzwerte der Eigenwerte, die wir im nächsten Unterkapitel betrachten.

	1	2	3	4	5	6
$\tilde{\omega}$	5.000000	10.000000	15.000000	20.000000	25.000000	30.000000
ω	4.057425	9.826058	15.956815	22.170349	15.956815	28.413934
ω^2	16.462695	96.551423	254.619961	491.524375	254.619961	807.351663

2.7 Verallgemeinertes Eigenwertproblem

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

Sei abermals $x_j := jh$, $j = 0, \ldots, n$ 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 $\mathbf{v}^{(n)} := (v(x_1), \ldots, v(x_{n-1}))^T$, bleiben ebenfalls nach wie vor so, wie sie waren.

$B_n\lambda$ soll nun, analog zu Vorher, κ^2 repräsentieren. Diesmal, ist κ jedoch als (stückweise konstante) Funktion zu verstehen. Also wird die Matrix B_n deren Fallunterscheidungen übernehmen und λ konstant bleiben. Es läuft darauf hinaus, dass

$$B_n := \begin{cases} \operatorname{diag}^{-2}(c_0, \ldots, c_0, c_1, \ldots, c_1,), & n-1 \in 2\mathbb{N} \\ \operatorname{diag}^{-2}(c_0, \ldots, c_0, \frac{c_0+c_1}{2}, c_1, \ldots, c_1,), & n-1 \in 2\mathbb{N}+1 \end{cases}, \qquad \lambda := \omega^2,$$

wobei c_0, c_1 in $B_n \in \mathrm{GL}_{n-1}(\mathbb{R})$ jeweils $\left\lfloor \frac{n-1}{2} \right\rfloor$-mal vorkommen. Dabei sei vorausgesetzt, dass $c_0, c_1 \neq 0$ und $c_0 + c_1 \neq 0$. Die Wahl von B_n lässt sich wie folgt begründen.

Seien $\mathbf{a}, \mathbf{b}$ Vektoren mit gleich vielen Komponenten. Dann ist die Matrix-Vektor-Multiplikation $\cdot$, mit einer erzeugten Diagonalmatrix, äquivalent zur komponentenweisen Multiplikaiton $\odot$.

$$\operatorname{diag}(\mathbf{a}) \cdot \mathbf{b} = \mathbf{a} \odot \mathbf{b} = \mathbf{b} \odot \mathbf{a} = \operatorname{diag}(\mathbf{b}) \cdot \mathbf{a} \tag{11}$$

Bei dem vorherigen Eigenwertproblem wäre B_n als Einheits-Matrix I_n zu interpretieren. Der Eigenwert λ konnte gleich ganz κ^2 approximieren, weil dieser Wert konstant war. Man hätte aber freilich auch mit der Skalarmatrix $(I_n c)^{-2}$ und ω^2 anstelle von κ^2 arbeiten können. Da κ^2 nun aber, als Funktion, zwei unterschiedliche Werte

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

annehmen kann, müssen wir die obere Eigenschaft (11) von Diagonalmatrizen ausnutzen. Damit realisieren wir die Fallunterscheidung zwischen $x_j < 1/2$ und $x_j > 1/2$. Für $x_j = 1/2$, was genau bei $n - 1 \in 2\mathbb{N}$ auftritt, wird gemittelt.

Nachdem die Inverse einer Diagonalmatrix genau die Matrix selbst mit komponentenweise Kehrwerten ist, lassen sich gleich B_n und B_n^{-1} leicht implementieren. Die zuständigen Funktionen besitzen die kreativen Namen `my_other_numpy_matrix` bzw. `my_other_numpy_matrix_inverse`.

```
def my_other_numpy_matrix_inverse(n, c):

    times = np.floor((n-1)/2)
    times = int(times)

    lower = [c[0]]*times
    upper = [c[1]]*times

    middle = [(c[0] + c[1])/2]

    if n%2 != 0:
        B_inverse = np.diag(lower + upper)**2
        return B_inverse
    else:
        B_inverse = np.diag(lower + middle + upper)**2
        return B_inverse

def my_other_numpy_matrix(n, c):
    return 1/my_other_numpy_matrix_inverse(n, c)
```

Das verallgemeinerte Eigenwertproblem $A_n\mathbf{v} = \lambda B_n\mathbf{v}$ werden wir zunächst auf $B_n^{-1}A_n\mathbf{v} = \lambda\mathbf{v}$ umformulieren. Dieses kann nun ebenfalls mit `np.linalg.eig`, für beliebige $n \geq 2$, gelöst werden.

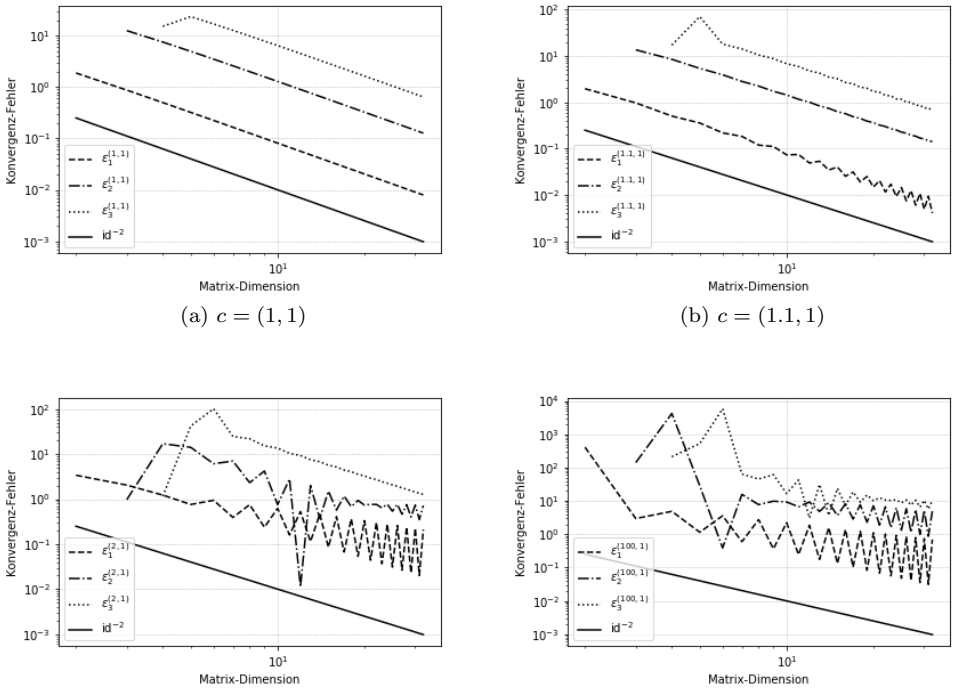
Die Matrix $B_n^{-1}A_n$ besitzt hoffentilch wieder $n - 1$ paarweise verschiedene Eigenwerte $\lambda^c_{1,n} < \cdots < \lambda^c_{n-1,n}$, die jeweils konvergieren.

Um einen ersten Eindruck des möglichen Konvergenz-Verhaltens zu bekommen, vergleichen wir die Eigenwerte mit den oberen semi-analytischen Ergebnissen mit $c_0 = 100$, $c_1 = 1$. Es folgt eine zur oberen analoge Tabelle mit Eigenwerten. Die Ergebnisse lassen zwar zu wünschen übrig, aber immerhin pendelt sich die Größenordnung rasch ein.

	2	3	4	5	6
1	20402.0	13.499662	21.329378	15.313793	20.048572
2	NaN	180004.500338	56813.374144	68.017688	96.940091
3	NaN	NaN	344805.296478	250012.501563	102552.962302
4	NaN	NaN	NaN	750004.166956	422576.319931

Wir bezeichnen (optimistischerweise) mit $\epsilon^c_i(n) := |\lambda^c_i - \lambda^c_{i,n}|$, $i = 1, \ldots, n-1$ den absoluten Konvergenz-

Fehler des i-ten Eigenwertes. λ_i^c erhalten wir durch ω^2 von `scipy.optimize.fsolve`, wobei $\tilde{\omega} := \sqrt{\lambda_{i,n}^c}$ als Startwert für das iterative Verfahren gewählt wird. Theoretisch hängt λ_i^c also noch von n ab. In der folgenden Abbildung 8 wurde ϵ_i^c mit der Vergleich-Geraden id^2, doppelt logarithmisch, geplottet.

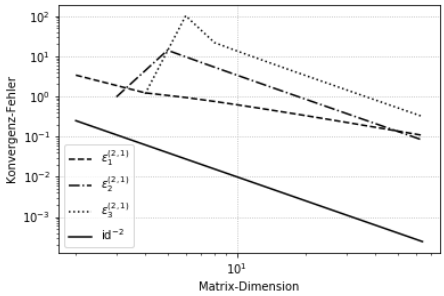


(a) $c = (1, 1)$ (b) $c = (1.1, 1)$

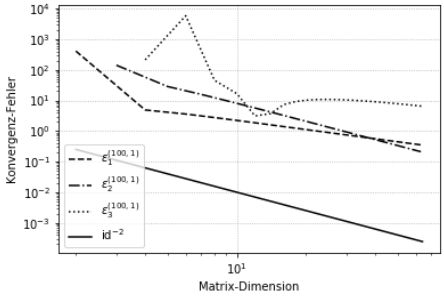
(c) $c = (2, 1)$ (d) $c = (100, 1)$

Abbildung 8: Konvergenz-Fehler der Eigenwerte von $B_n^{-1}A_n$, für $n = 2, \ldots, 32$ und

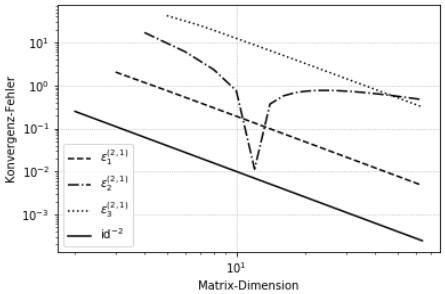
Es macht Sinn, dass Abbildung 8a mit Abbildung 5 korreliert. Diese „Bergsteiger-Konvergenz“ steigt anscheinend mit dem Verhältnis c_0/c_1 (no pun intended). Etwas Aufschlussreicher werden 8c und 8d, wenn man gerade und ungerade n unterscheidet. Diese Abbildung 8 bleibt übrigens unverändert, wenn man die Komponenten von c vertauscht.



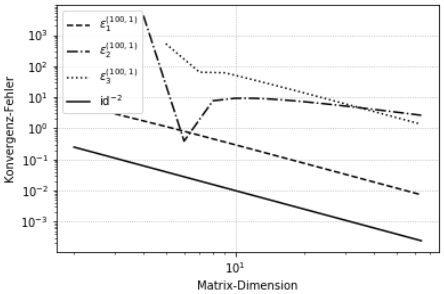
(a) $c = (2, 1)$, $n \in 2\mathbb{N}$



(b) $c = (100, 1)$, $n \in 2\mathbb{N}$



(c) $c = (2, 1)$, $n \in 2\mathbb{N} + 1$



(d) $c = (100, 1)$, $n \in 2\mathbb{N} + 1$

Abbildung 9: Konvergenz-Fehler der Eigenwerte von $B_n^{-1}A_n$, für $n = 2, \ldots, 64$ und

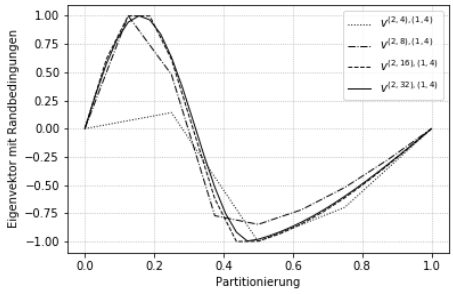
Wenn man für $c = (100, 1)$ gerade und ungerade betrachtet, so bemerkt man, dass $\epsilon_2^{(100,1)}$ bei Abbildung 9b konvergiert und $\epsilon_1^{(100,1)}, \epsilon_3^{(100,1)}$ bei Abbildung 9d. Dies lässt vermuten, dass im Allgemeinen die besten Approximationsstrategie ist, Teifolgen zu betrachten.

$$\text{Benutze } \begin{cases} (\lambda_{i,n}^c)_{n\in 2\mathbb{N}} & \text{für } i \in 2\mathbb{N}, \\ (\lambda_{i,n}^c)_{n\in 2\mathbb{N}+1} & \text{für } i \in 2\mathbb{N}+1. \end{cases}$$

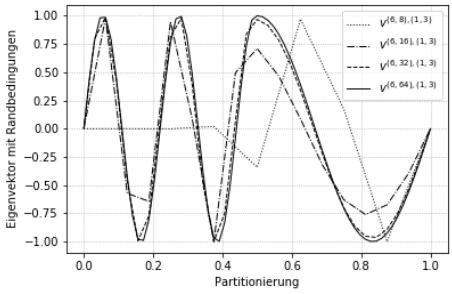
Seien $\mathbf{v}^{(1,n),c}, \ldots, \mathbf{v}^{(n-1,n),c}$ die Eigenvektoren (modulo Konstante), zu den Eigenwerten $\lambda_{1,n}^c < \cdots < \lambda_{n-1,n}^c$, der Matrix $B_n^{-1}A_n$. Wir antizipieren ein weniger schönes Konvergenz-Verhalten, als das der Eigenvektoren der Matrizen $(A_n)_{n\in\mathbb{N}}$. Nichts desto trotz, wurden die Eigenvektoren $\mathbf{v}^{(i,n_i),c}$, normiert bzgl. $\|\cdot\|_\infty$, geplottet, wobei

$$\begin{aligned} c_{\max} &= 4, & c &\in \{(c_0, c_1) : c_0, c_1 = 1, \ldots, c_{\max}\}, \\ i &= 1, \ldots, 2c_{\max}, & n_i &\in \left\{2^{p_{\min}+p_{\text{add}}} : p_{\min} = \lceil \log_2(i+1) \rceil, \, p_{\text{add}} = 0, \ldots, 3\right\} =: N_i. \end{aligned}$$

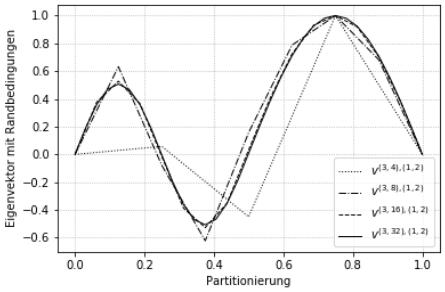
Dabei wurden die Vektoren $\left\{\mathbf{v}^{(i,n),c} : n \in N_i\right\}$ jeweils zu einem Bild zusammengefasst. Nachdem wir dadurch auf 128 Bilder kommen, werden wir nicht alle herzeigen. Außerdem, sieht nur ein Bruchteil davon „schön“ aus.



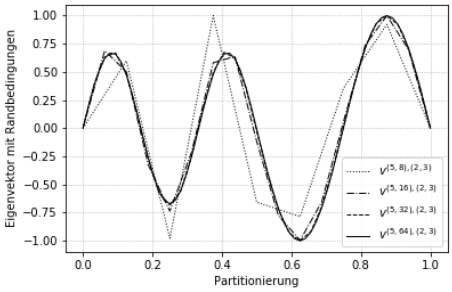
(a) mit $i = 2$, $n = 4, 8, 16, 32$, und $c = (1, 4)$



(b) mit $i = 6$, $n = 8, 16, 32, 64$, und $c = (1, 3)$



(c) mit $i = 3$, $n = 4, 8, 16, 32$, und $c = (1, 2)$



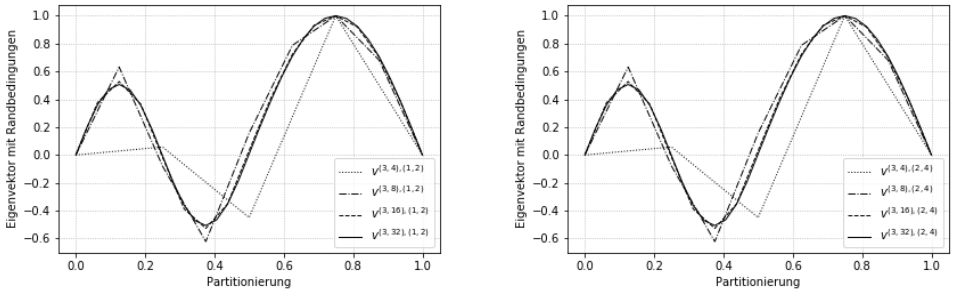
(d) mit $i = 5$, $n = 8, 16, 32, 64$, und $c = (2, 3)$

Abbildung 10: Eigenvektoren $\mathbf{v}^{(i,n),c}$ der Matrizen $B_n^{-1} A_n$

Mit „schön“ ist gemeint, dass die Grenzfunktionen bzgl. n der Eigenvektoren $\mathbf{v}^{(i,n),c}$ seinen letzte (halbe) Schwingungs-Periode vollenden kann, bevor die nächste Ausbreitungsgeschwindigkeit übernimmt. Mit anderen Worten, die beiden Funktionshälften treffen sich an der x-Achse. Man fragt sich nun vielleicht, für welche Ausbreitungsgeschwindigkeiten c und Eigenpaar-Nummerierung i diese Grenzfunktionen „schön“ aussehen.

Dazu bemerken wir zuerst, dass die selben Plots herauskommen, wenn $c^{(1)}, c^{(2)}$ bis auf eine Konstante übereinstimmen. Wegen der Normierung bzgl. $\|\cdot\|_\infty$, sieht man das durch hinschauen (auf $A\mathbf{v} = \lambda B\mathbf{v}$), also eigentilch auch bereits a priori.

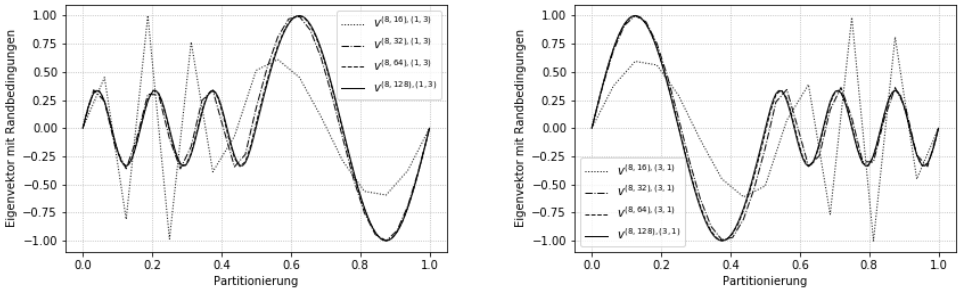
$$\mathbf{v}^{(i,n),c^{(1)}} = \mathbf{v}^{(i,n),c^{(2)}}, \text{ für } c^{(1)} \equiv c^{(2)} \bmod \text{Konstante}.$$



(a) $c = (1, 2)$ (b) $c = (2, 4)$

Abbildung 11: Eigenvektoren $\mathbf{v}^{(3,n),c}$ der Matrizen $B_n^{-1}A_n$, mit $n = 8, 16, 32, 64$ und

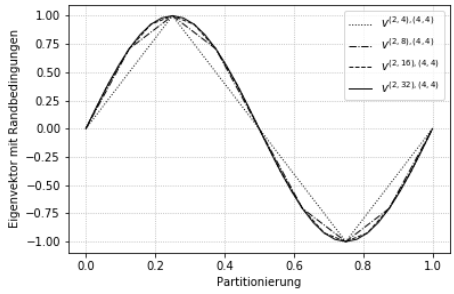
A posteriori hingegen, bemerken wir, dass zwei Plots mit (c_0, c_1) bzw. (c_1, c_0) auch graphisch zusammenhängen. Die erste und zweite Hälft der Grenzfunktionen tauschen, wenn man zwischen den Plots wechselt.



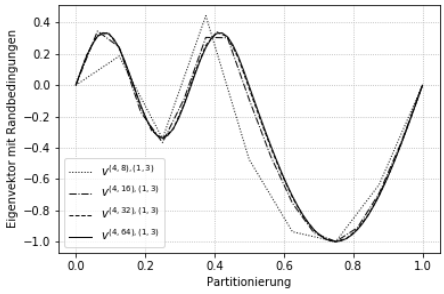
(a) $c = (1, 3)$ (b) $c = (3, 1)$

Abbildung 12: Eigenvektoren $\mathbf{v}^{(8,n),c}$ der Matrizen $B_n^{-1}A_n$, mit $n = 16, 32, 64, 128$ und

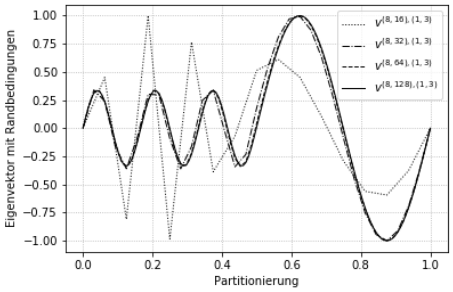
Es sticht aber noch etwas Anderes ins Auge. Wenn man irgendeinen dieser Plots betrachtet, dann geht die Grenzfunktion genau dann durch den ausgezeichneten Punkt $(1/2, 0)$, wenn $\tilde{c_0} + \tilde{c_1} \,|\, i$, wobei $\tilde{c_0} : \tilde{c_1} = c_0 : c_1$ und $\tilde{c}$ vollständig gekürzt ist.



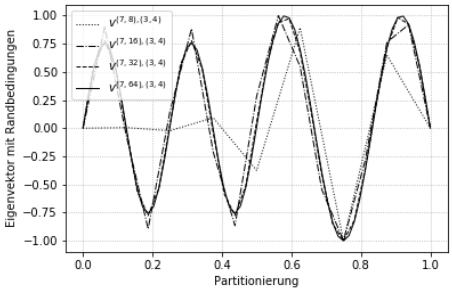
(a) mit $i = 2$, $n = 4, 8, 16, 32$, und $c = (4, 4)$



(b) mit $i = 4$, $n = 8, 16, 32, 64$, und $c = (1, 3)$



(c) mit $i = 8$, $n = 8, 16, 32, 64$, und $c = (1, 3)$



(d) mit $i = 7$, $n = 8, 16, 32, 64$, und $c = (3, 4)$

Abbildung 13: Eigenvektoren $\mathbf{v}^{(i,n),c}$ der Matrizen $B_n^{-1} A_n$

Das kann man heuristisch so begründen, dass die Ausbreitungsgeschwindigkeiten c, gemeinsam mit i, für die relativen Anzahlen der Schwingungsperioden verantwortlich sind; d.h, diese stehen diese im reziproken Verhältnis zu einander. Für $i = 4$, $c = (1, 3)$ aus Abbildung 13b zum Beispiel, schwingt die erste Hälfte doppelt so schnell, wie die zweite. Dasselbe geschieht für $i = 8$, wobei die Grenzfunktion als Ganzes doppelt so schnell schwingt, wie bei $i = 4$. Durch dieses Zwischenspiel von c und i, finden wir die meisten „schönen“ Funktionen.

Was ist aber mit dem „künstlich verallgemeinerten“ Fall $c = (1, 1) = \cdots = (c_{\max}, c_{\max})$, $i \in \mathbb{N} - 1$? Wieso sieht der so „schön“ aus? Um eine Erklärung zu finden, verallgemeinern wir fröhlich weiter und betrachten die Material-Funktion $c = (c_0, \ldots, c_m) \in (\mathbb{Q}^+)^{m+1}$, wobei diese, analog zu (9), bzgl. $1/(m + 1)$ äquidistant zu verstehen ist.

$$c(x) := \begin{cases} c_0, & x \in (0, 1/(m+1)) \\ \vdots & \vdots \\ c_m, & x \in (m/(m+1), 1) \end{cases}$$

Wir stellen fest, dass sich die naheliegende Verallgemeinerung der oberen Regel für diese Fälle gilt. Die Grenzfunktion geht genau dann durch die ausgezeichneten äquidistanten Punkte $\{k/(m+1)\}_{k=0}^{m+1}$, wenn $|\tilde{c}| := \tilde{c_0} + \cdots + \tilde{c_m} \mid i$, wobei $\tilde{c_0} : \cdots : \tilde{c_m} = c_0 : \cdots : c_m$ und $\tilde{c}$ als Ganzes vollständig gekürzt ist.

Tatsächlich gibt es im „künstlich verallgemeinerten“ Fall mit beliebigem i ein m, sodass für $c \in (\mathbb{Q}^+)^{m+1}$ gilt $|\tilde{c}| = m + 1 \mid i$. Dieser Fall liefert übrigens, wegen der bereits angesprochenen Skalarmatrix, immer dasselbe Resultat.

2.8 Vektor-Iteration

Sei $\rho > 0$. Wir Verwenden die Vektor-Iteration angewendet auf das Eigenwertproblem

$$(A - \rho B)^{-1} B\tilde{\mathbf{v}} = \mu\tilde{\mathbf{v}} \tag{12}$$

mit den Matrizen A und B aus dem vorigen Unterkapitel.

Das macht man, weil die Vektor-Iteration in ihrer einfachsten Form, `vector_iteration_simple`, nur den Eigenvektor mit dem betragsgrößten Eigenwert liefert.

```
# m ... number of iterations

def vector_iteration_simple(M, m):

    # some random (non zero) vector
    randy = np.random.rand(M.shape[0])
    # normalize
    randy = randy/np.linalg.norm(randy)

    for _ in range(m):

        # apply matrix
        randy = M @ randy
        # normalize
        randy = randy/np.linalg.norm(randy)

    return randy
```

Es besteht ein Zusammenhang zwischen den Eigenpaaren $(\mu, \tilde{\mathbf{v}})$ und den Eigenpaaren $(\lambda, \mathbf{v})$ aus der vorigen Aufgabe. Für $\mu \neq 0$, ist dieser $\lambda = \rho + \frac{1}{\mu}$, weil dann

$$A\mathbf{v} = \lambda B\mathbf{v} \Leftrightarrow A\mathbf{v} = \left(\rho + \frac{1}{\mu}\right) B\mathbf{v} \Leftrightarrow B\mathbf{v} = \mu(A - \rho B)\mathbf{v} \Leftrightarrow (A - \rho B)^{-1} B\mathbf{v} = \mu\mathbf{v}.$$

Bei der Vektor-Iteration wird der potentielle Eigenvektor $\mathbf{v}$ ständig normiert. Damit kann man wegen $A\mathbf{v} = \lambda\mathbf{v}$ sehr rasch auf den zugehörigen Eigenwert λ kommen.

$$\langle A\mathbf{v}, \mathbf{v}\rangle = \langle \lambda\mathbf{v}, \mathbf{v}\rangle = \lambda\|\mathbf{v}\|^2 = \lambda$$

Jetzt können wir auch ein schlauereres Abbruch-Kriterium wählen. Die Vektor-Iteration soll terminieren, wenn die Änderung der Eigenwerte hinreichend klein ist. Die Implementierung, die diese Überlegungen beherzigt, folgt mit `vector_iteration_unshifted`.

```
def vector_iteration_unshifted(M, tol):

    # some random (non zero) vector
    randy = np.random.rand(M.shape[0])
    # normalize
    randy = randy/np.linalg.norm(randy)

    # stop iteration when differences are small enough
    eigen_randy_old = -tol
    eigen_randy_new = tol

```

```
    while abs(eigen_randy_old - eigen_randy_new) > tol:

        # get eigen values of randy
        eigen_randy_old = np.dot(M @ randy, randy)

        # apply matrix
        randy = M @ randy
        # normalize
        randy = randy/np.linalg.norm(randy)

        # get eigen values of randy
        eigen_randy_new = np.dot(M @ randy, randy)

    # use best approximation of eigen value
    eigen_randy = eigen_randy_new

    # get eigen pair
    eigen_pair = (eigen_randy, randy)

    return eigen_pair
```

Die untere Implementierung, `vector_iteration_shifted`, der Vektor-Iteration des geshifteten Eigenwertproblems (12), ist ein gutes Beispiel eines sinnvollen Einsatzes der LU-Zerlegung. Diese ist zwar aufwändig, aber nachdem sie in jedem Iterationsschritt verwendet werden kann, zahlt sich dieser einmalige Aufwand aus.

```
from scipy.linalg import lu, solve_triangular

def vector_iteration_shifted(n, c, rho, tol):

    # some random (non zero) vector
    randy = np.random.rand(n-1)
    # normalize
    randy = randy/np.linalg.norm(randy)

    # get matrices
    A = my_numpy_matrix(n)
    B = my_other_numpy_matrix(n, c)
    B_inverse = my_other_numpy_matrix_inverse(n, c)
    B_inverse_A = B_inverse @ A

    # calculate lu-decomposition and apply permutation matrix
    M = A - rho*B
    P, L, U = lu(M)

    # used to get eigen value of randy
    M = np.linalg.inv(A - rho*B) @ B

    # stop iteration when differences are small enough
    eigen_randy_old = -tol
    eigen_randy_new = tol

    while abs(eigen_randy_old - eigen_randy_new) > tol:

        # get eigen values of randy
        eigen_randy_old = np.dot(M @ randy, randy)

        # apply first part of matrix to randy
        randy = P @ B @ randy

        # solve L @ forwards = randy via forwards substitution
        forwards  = solve_triangular(L, randy,    lower = True)
        # solve U @ backwards = forwards per backwards substitution
        backwards = solve_triangular(U, forwards, lower = False)
```

```
39
40          # apply second part of matrix to randy
41          randy = backwards
42          # normalize
43          randy = randy/np.linalg.norm(randy)
44
45          # get eigen values of randy
46          eigen_randy_new = np.dot(M @ randy, randy)
47
48      # use best approximation of eigen value
49      eigen_randy = eigen_randy_new
50
51      # get eigen pair of shifted problem
52      eigen_pair_shifted = (eigen_randy, randy)
53
54      # get eigen pair of unshifted problem
55      eigen_randy_unshifted = 1/eigen_randy + rho
56      eigen_pair_unshifted  = (eigen_randy_unshifted, randy)
57
58      return eigen_pair_shifted, eigen_pair_unshifted
```

Angenommen, wir wüssten bereits, dass all unsere Eigenwerte $\lambda^c_{1,n} < \cdots < \lambda^c_{n-1,n} \in \mathbb{R}^+$ reell und positiv sind. `vector_iteration_simple` liefert uns den Eigenvektor mit dem betragsgrößten Eigenwert und `vector_iteration_shifted` den Eigenvektor mit dem Eigenwert, der am nähesten bei ρ liegt. Nun können wir mit einer binären Suche, ähnlich zum Bisektionsverfahren, alle Eigenwerte finden und ein eigenes `np.linalg.eig` programmieren.

```
1  from timeit import default_timer as timer
2
3  def recursion(n, c, tol, eigen_pairs, bound_lower, bound_upper):
4
5      bound_middle = rho = (bound_lower + bound_upper)/2
6
7      print("searching for eigen pair with eigen value near", rho, "...")
8      start = timer()
9      eigen_pair_middle = vector_iteration_shifted(n, c, rho, tol)[1]
10     end = timer()
11     bound_middle = eigen_pair_middle[0]
12     print("found eigen value", bound_middle, "in", end-start, "seconds", "\n")
13
14     if abs(bound_middle - bound_lower) < 10 or bound_middle <= bound_lower:
15         print("bound_lower ~ bound_middle")
16         print(bound_lower, "~", bound_middle)
17         print("or")
18         print("bound_middle <= bound_lower")
19         print(bound_middle, "<=", bound_lower)
20         print("stopping persuit", "\n")
21         return False
22     if abs(bound_upper - bound_middle) < 10 or bound_upper <= bound_middle:
23         print("bound_middle ~ bound_upper")
24         print(bound_middle, "~", bound_upper)
25         print("or")
26         print("bound_upper <= bound_middle")
27         print(bound_upper, "<=", bound_middle)
28         print("stopping persuit", "\n")
29         return False
30
31     eigen_pairs.update([eigen_pair_middle])
32     print("eigen_pair_middle (new one) =", eigen_pair_middle, "\n")
33
34     if len(eigen_pairs) == n-1:
35         print("len(eigen_pairs) == n-1")
```

```
        print(len(eigen_pairs), "=", n-1)
        print("All eigen pairs found", "\n")
        return True

    print("repeating recursion with:")
    print("bound_lower =", bound_lower)
    print("bound_middle =", bound_middle, "\n")
    recursion(n, c, tol, eigen_pairs, bound_lower, bound_middle)

    if len(eigen_pairs) == n-1:
        print("len(eigen_pairs) == n-1")
        print(len(eigen_pairs), "=", n-1)
        print("All eigen pairs found", "\n")
        return True

    print("repeating recursion with:")
    print("bound_middle =", bound_middle)
    print("bound_upper =", bound_upper, "\n")
    recursion(n, c, tol, eigen_pairs, bound_middle, bound_upper)

    return True

def get_my_eigen_pairs_with_vector_iteration(n, c, tol):

    print("Starting search with n =", n, "...", "\n")

    B_inverse_A = my_general_numpy_matrix(n, c)
    eigen_pairs = {}

    rho = 0

    eigen_pair_upper = vector_iteration_unshifted(B_inverse_A, tol)
    bound_upper = eigen_pair_upper[0]

    eigen_pairs.update([eigen_pair_upper])
    print("eigen_pair_upper =", eigen_pair_upper, "\n")

    if len(eigen_pairs) == n-1:
        print("len(eigen_pairs) == n-1")
        print(len(eigen_pairs), "=", n-1)
        print("All eigen pairs found", "\n")
        return eigen_pairs

    eigen_pair_lower = vector_iteration_shifted(n, c, rho, tol)[1]
    bound_lower = eigen_pair_lower[0]

    eigen_pairs.update([eigen_pair_lower])
    print("eigen_pair_lower =", eigen_pair_lower, "\n")

    if len(eigen_pairs) == n-1:
        print("len(eigen_pairs) == n-1")
        print(len(eigen_pairs), "=", n-1)
        print("All eigen pairs found", "\n")
        return eigen_pairs


    print("bound_lower =", bound_lower)
    print("bound_upper =", bound_upper, "\n")
    recursion(n, c, tol, eigen_pairs, bound_lower, bound_upper)

    return eigen_pairs
```

Das Abbruch-Kriterium ist aber anscheinend noch immer suboptimal, weil die damit berechneten Eigenwerte, sogar für `tol = 1e-15`, nicht ganz denen von `np.linalg.eig` entsprechen.

	2	3	4	5	6
1	20402.0	13.499662	21.329378	15.313793	20.048572
2	NaN	180004.500338	56813.374137	68.003831	96.940090
3	NaN	NaN	344805.296478	180595.546136	102552.962297
4	NaN	NaN	NaN	750004.166956	392219.488916

Zum Vergleich, zeigen wir nochmal die Eigenwerte via `np.linalg.eig`. Betrachtet man die Eigenwerte $\lambda_{3,5}^{(100,1)}, \lambda_{4,6}^{(100,1)}$, so merkt man, dass Ungenauigkeiten beim vorletzten Eigenpaar, d.h. Eigenpaar mit betragsmäßig zweitgrößten Eigenwert, auftreten, wenn n groß ist. Offensichtlich, lässt sich diese Eigenwertsuche noch optimieren, das würde aber den Rahmen dieses Projektes sprengen.

	2	3	4	5	6
1	20402.0	13.499662	21.329378	15.313793	20.048572
2	NaN	180004.500338	56813.374144	68.017688	96.940091
3	NaN	NaN	344805.296478	250012.501563	102552.962302
4	NaN	NaN	NaN	750004.166956	422576.319931

Zuletzt, wollen wir noch wissen, ob die Eigenwerte unserer Matrix $B_n^{-1}A_n$ tatsächlich alle positiv und reell sind. Sonst wäre der Algorithmus ja, aus mathematische Sicht, wertlos. Eine Möglichkeit bietet die explizite Darstellung von Eigenpaaren von Tridiagonalmatrizen. Wir wissen aber, dass das Gewünschte genau dann gilt, wenn $B_n^{-1}A_n$ positiv definit ist. Das können wir mit dem Hauptminoren-Kriterium locker überprüfen.

$$A'_n := \begin{pmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix}$$

Zuerst berechnen wir die Determinante der Matrix A'_n. Dazu induzieren wir $\det(A'_n) = n$. Der Induktionsanfang, $n = 2, 3$, ist trivial. Für den Induktionsschritt muss man einmal nach der letzten Spalte und dann Zeile entwickeln (oder umgekehrt).

Nun gilt aber $A_n = \frac{1}{h^2}A'_n$ und $B_n^{-1} > 0$ ist eine Diagonalmatrix mit positiven Einträgen. Weil die Determinante solcher Matrizen bloß das Produkt ihrer Komponenten ist, sind alle Hauptminoren unserer Matrizen positiv. Sie sind also tatsächlich positiv definit. Deren Produkt und Vielfaches A_n ist es also auch, weil

$$0 < \mathbf{x}^* A\mathbf{x}, \mathbf{x}^* B\mathbf{x} \Rightarrow 0 < \mathbf{x}^* A\mathbf{x}\mathbf{x}^* B\mathbf{x} = \mathbf{x}^* A\langle \mathbf{x}, \mathbf{x}\rangle B\mathbf{x} = \mathbf{x}^* A\|\mathbf{x}\|^2 B\mathbf{x} \Rightarrow 0 < \mathbf{x}^* AB\mathbf{x}.$$

3 Titel