Angewandte Mathematik
Modellierung und Numerik
Daniel Tenbrinck, Tim Roith, Lea Föcke

Friedrich-Alexander-Universität
Naturwissenschaftliche Fakultät

Einführung in die Numerik

Programmieraufgaben Blatt 02, Abgabe: Mittwoch, 23.11.2022, 12:00
Wintersemester 2022/2023

— AUFGABE 1: Lösen einer DGL; 2 + 4 + 3 + 3 Punkte —

Wir betrachten die eindimensionale *Poisson-Gleichung* auf dem Intervall $I = [a, b]$,

$$\begin{aligned} -\Delta u(x) &= f(x), \quad x \in (a, b), \\ u(a) &= u_a, \\ u(b) &= u_b, \end{aligned}$$

mit gegebener Funktion $f : I \to \mathbb{R}$ und gegebenen Werten $u_a, u_b \in \mathbb{R}$. Hierbei ist der Laplace-Operator definiert durch $\Delta u := u''$. Realisieren Sie folgende Schritte in Python.

(i) Erstellen Sie eine Klasse `Simple_DGL`, welche die Poisson-Gleichung mit Lösung

$$v(x) = x^4 - 3x^3 + 2x^2 + 1.$$

repräsentieren soll.

- Schreiben Sie hierfür eine Methode `__init__(self, a=0., b=1.)`, s.d. die Attribute `a` und `b` die Intervallgrenzen speichern.
- Weiterhin soll die Klasse eine Methode `__call__(self, x)` haben, welche für einen Input `x` die Auswertung von v an diesem Punkt ausgibt.
  Hinweis: Diese Methode repräsentiert die analytische Lösung, die man in der Realität eigentlich nicht gegeben hat. Wir konstruieren uns hier ein Beispiel in welchem wir die Lösung kennen.
- Implementieren Sie eine Methode `boundary(self)` welche den Vektor $[v(a), v(b)]$ zurückgibt.
  Hinweis: Auch diese Methode wäre in echten Anwendungen anders gegeben. Wir wählen hier gerade die Auswertung von v damit wir auch v als Lösung erhalten.
- Implementieren sie zusätzlich eine Methode `rhs(self, x)`, welche die rechte Seite f in der DGL repräsentiert. Da wir v als Lösung erhalten wollen, geben wir hier $-\Delta v(\mathtt{x})$ aus.
  Hinweis: Berechnen Sie die zweite Ableitung analytisch.

(ii) Auf Gittern mit $N \in \mathbb{N}$ äquidistanten Punkten

$$\begin{aligned} x_i &:= h \cdot i + a, \quad i = 0, \ldots, N \\ h &:= \frac{b-a}{N} \end{aligned}$$

mit kann der eindimensionale Laplace-Operator durch folgende Tridiagonalmatrix diskretisiert werden:

$$-\Delta \approx A_h = \frac{1}{h^2} \begin{pmatrix} 2 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \vdots \\ 0 & \ddots & \ddots & \ddots & \dots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{pmatrix} \in \mathbb{R}^{(N-1)\times(N-1)}.$$

Um die Gleichung approximativ zu lösen betrachten wir für

$$F_h \in \mathbb{R}^{N-1}, \quad (F_h)_i = f(x_i)$$

das Gleichungssystem

$$A_h \tilde{U}_h = F_h + G_h \tag{1}$$

wobei der Vektor $\tilde{U}_h \in \mathbb{R}^{N-1}$ die echte Lösung u an den Gitterpunkten approximieren soll, d.h.,

$$(\tilde{U}_h)_i \approx u(x_i), \quad i = 1, \dots, N-1.$$

Der Vektor G_h soll die Randbedingungen der Differentialgleichung korrekt einbeziehen. Der Lösungsvektor $U_h \in \mathbb{R}^{N+1}$ ergibt sich dann durch das Hinzufügen der Randwerte, d.h.,

$$(U_h)_0 := u_a, \quad (U_h)_i := (\tilde{U}_h)_i, \quad i = 1, \dots, N-1, \quad (U_h)_N := u_b.$$

Implementieren Sie eine Funktion `solvePoisson(DGL, N=50, solver='Gauss')`, welche die Poisson-Gleichung approximativ löst. Hierbei ist `DGL` eine Klasse, welche wie in (i) eine DGL repräsentiert und N gibt die Anzahl der Gitterpunkte an. Die Funktion soll U_h zurückgeben. Die Option `solver` soll hierbei spezifizieren welche Methode benutzt wird um das LGS in Gleichung (1) zu lösen. Implementieren Sie die Optionen

- `'Gauss'`: Lösen des LGS mit dem Gauß-Algorithmus von Blatt 1,
- `'Cholesky'`: Lösen des LGS mithilfe der Cholesky Implementierung von Tutorium 2,
- `'numpy'`: Lösen des LGS mithilfe von `numpy.linalg.solve`.

und eine Fehlermeldung falls eine unbekannte Option spezifiziert wird.

(iii) Plotten sie die berechneten Lösungen für $N = 4, 8, 16, 32$ und vergleichen Sie sie mit der exakten Lösung.

(iv) Vergleichen Sie die Laufzeiten für $N = 2, 4, 8, 16, 32, 64, 128$ für alle drei Versionen `'Gauss'`, `'Cholesky'`, `'numpy'`.

— AUFGABE 2: Bildkompression mit Singulärwertzerlegung; 4 + 4 + 4 Punkte

Es sei $A \in \mathbb{R}^{m\times n}$ eine Matrix, wir betrachten in dieser Aufgabe Varianten der Singulärwertzerlegung (im Englischen: Singular value decomposition (SVD)), welche den Speicherplatz verringern. Benutzen Sie hierfür im folgenden die Funktion `np.linalg.svd`.

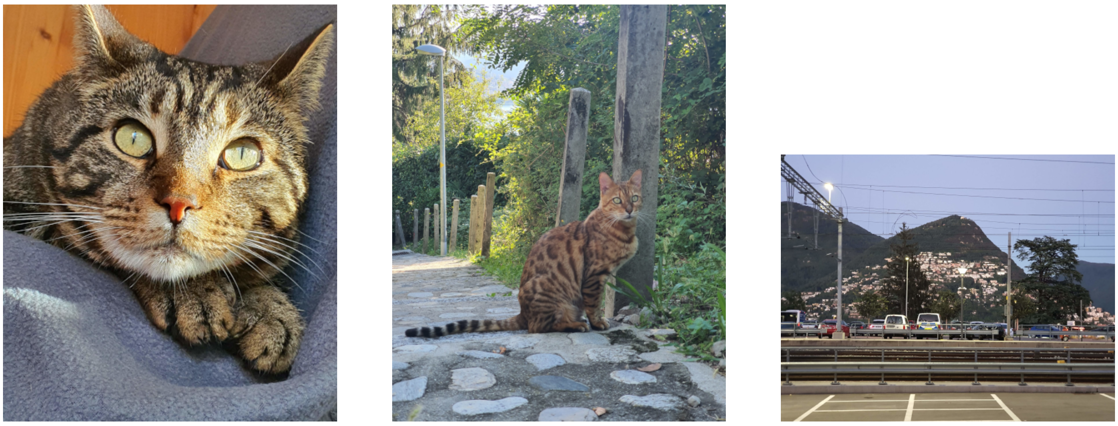
(i) Schreiben Sie eine Funktion `svd(A, variant='standard', t=None)` welche die folgenden Varianten implementiert.

- `variant='standard'`: In diesem Fall soll die Standard-Variante der SVD ausgeben werden, d.h. für $U \in \mathbb{R}^{m\times m}, V \in \mathbb{R}^{n\times n}, \Sigma \in \mathbb{R}^{m\times n}$ mit
$$A = U\Sigma V^T$$
sollen U, Σ, V ausgeben werden.
Hinweis: Achten Sie darauf, falls Sie `np.linalg.svd` benutzen, dass die Matrizen die Sie hier zurückgeben wirklich die richtige Form haben!
- `variant='truncated'`: Für $\mathbb{N} \ni t \leq \min(m, n)$ ist die *abgeschnittene* SVD definiert durch
$$A \approx \tilde{A} = U_t\Sigma_t V_t^T$$
wobei $U_t \in \mathbb{R}^{m\times t}, V_t \in \mathbb{R}^{n\times t}$ jeweils die ersten t Spalten von U, V enthalten und $\Sigma \in \mathbb{R}^{t\times t}$ die obere linke $k \times k$ Untermatrix von Σ ist. Geben Sie hier $U_t, \tilde{\Sigma}_t, V_t$ aus, wobei $\tilde{\Sigma} \in \mathbb{R}^t$ die Diagonale von Σ bezeichnet.
Hinweis: Wir gehen hier und im Folgenden davon aus, dass die Singulärwerte absteigend in Σ gespeichert sind, s.d. wir die t größten Werte behalten.
- `variant='thin'`: Es sei $k = \min(m, n)$ dann ist die *dünne* SVD definiert als die abgeschnittene SVD für $t = k$.
- `variant='compact'`: Es sei $r \leq \min(m, n)$ die Anzahl der positiven Singulärwerte. Die *kompakte* SVD ist dann definiert als die abgeschnittene SVD für $t = r$.

(ii) In dieser Aufgaben wollen wir den Speicheraufwand für Bilder mithilfe der SVD verringern. Schreiben Sie hierzu eine Klasse `compressed_img` welche ein Bild in komprimiertem Speicherformat repräsentieren soll.

- Die Methode `__init__(self, I, factor=1.0)` erhält ein Bild als `numpy` Array `I`$\in \mathbb{R}^{N_h\times N_w\times N_c}$ wobei
  - N_h : Höhe des Bildes,
  - N_w : Breite des Bildes,
  - N_c : Anzahl der Farb-Channel (3 für RGB Bilder + 1 für Transparenz).

  Der Wert `factor` stellt den Kompressionsfaktor dar. Die Klasse soll **nicht** das Bild `I` speichern sondern lediglich Matrizen `U`$\in \mathbb{R}^{N_h\times t\times N_c}$, `S`$\in \mathbb{R}^{t\times N_c}$, `V`$\in \mathbb{R}^{N_w\times t\times N_c}$ welche mit der nachfolgenden Methode `compress_svd` erzeugt werden. Der Wert t soll hier so gewählt werden, dass für



(a) Bild `cat-1.png`. (b) Bild `cat-2.png`. (c) Bild `Lugano.png`.

Abbildung 1: Bilder für Aufgabe 2.

- die Anzahl der Einträge im komprimierten Format $N_{\text{SVD}} = N_c \cdot (N_h \cdot t + t + N_w \cdot t)$
- und die Anzahl der Einträge im originalen Bild $N_I = N_h \cdot N_w \cdot N_c$

gilt,

$$\frac{N_{\text{SVD}}}{N_I} \approx \texttt{factor}.$$

- Implementieren Sie die Methode `compress_svd(self, I, t)` welche für jeden Channel `I[:,:,i]`$\in \mathbb{R}^{N_h \times N_w}$ die abgeschnittene SVD für ein t berechnet und in Arrays `U`$\in \mathbb{R}^{N_h \times t \times N_c}$, `S`$\in \mathbb{R}^{t \times N_c}$, `V`$\in \mathbb{R}^{N_w, t, N_c}$ speichert. Diese Arrays sollen als Attribute der Klasse gesetzt werden.
- Implementieren Sie die Methode `__call__(self)` welche bei Aufruf das komprimierte Bild zurückgibt, indem die Matrizen `U,S,V` passend aneinander multiplizieren.

(iii) Testen Sie ihre Klasse anhand der drei gegebenen Bildern in Abb. 1. Visualisieren Sie jeweils das komprimierte Bild und geben Sie den Fehler zum originalen Bild aus.

Hinweis: Um Bilder als `numpy` Array einzulesen, können Sie die Methode `matplotlib.pyplot.imread` verwenden. Um Bilder zu visualisieren können Sie `matplotlib.pyplot.imshow` benutzen.