Aufgabe2

October 26, 2017

1 Aufgabe 2

1.1 Ziel

Lösen Sie das Randwertproblem für die gewöhnliche Differentialgleichung

$$\begin{aligned} -u''(x) &= f(x) \quad \text{für } x \in (0,1), \\ u(x) &= g(x) \quad \text{für } x \in \{0,1\}. \end{aligned}$$

mit Hilfe eines Finite-Differenzen Verfahrens. Als Testproblem soll $f(x) = sin(2\pi x)$ und $g(x) = 0$ dienen.

Zu gegebenem $N \in \mathbb{N}$ definieren wir $h = \frac{1}{N}$ und die Stützstellen $x_i = i\,h$, $i = 0, \ldots, N$ und approximieren

$$\begin{aligned} u(x_i) &\approx u_i && i = 0, \ldots, N, \\ f(x_i) &\approx f_i && i = 0, \ldots, N, \\ u''(x_i) &\approx \frac{1}{h^2}(u_{i-1} - 2u_i + u_{i+1}) && i = 1, \ldots, N-1. \end{aligned}$$

Verwenden Sie ein DUNE Gitter (`onedGrid`), um das das Gleichungssystem zu assemblieren, lösen Sie es mit Hilfe des CG Verfahrens und visualisieren Sie das Ergebnis für verschiedene Gitterweiten.

Ferner werde mit u_h die stückweise lineare Interpolierende der u_i, d.h. $u_h(x_i) = u_i$ bezeichnet und sei e_h der L^2-Fehler

$$e_h^2 := \int_0^1 \|u - u_h\|^2.$$

Verwenden Sie eine DUNE Quadratur, um dieses Integral zu approximieren. Berechnen Sie für die angegebenen Werte von N den Fehler e_h die experimentelle Konvergenzordnung.

1.2 Beispiel: Ein DUNE Gitter

Legen wir exemplarisch ein `onedGrid` mit konstanter Gitterweite:

```
In [1]: from dune.grid import cartesianDomain, onedGrid

        grid = onedGrid(cartesianDomain([0], [1], [4]))
```

Wir können über die Elemente (`elements`) und Knoten (`vertices`) eines Gitters iterieren:

```
In [2]: barys = [str(e.geometry.center) for e in grid.elements()]
        print("Schwerpunkte der Elemente: " + ", ".join(barys))

        vertices = [str(v.geometry.center) for v in grid.vertices()]
        print("Position der Knoten: " + ", ".join(vertices))

        volumes = [str(e.geometry.volume) for e in grid.elements()]
        print("Volumina der Elemente: " + ", ".join(volumes))
```

```
Schwerpunkte der Elemente: (0.125000), (0.375000), (0.625000), (0.875000)
Position der Knoten: (0.000000), (0.250000), (0.500000), (0.750000), (1.000000)
Volumina der Elemente: 0.25, 0.25, 0.25, 0.25
```

Um Daten an das Gitter anzuheften, stellt uns das Gitter einen `indexSet` zur Verfügung. Dieser ordnet jeder Entität im Gitter einen Index $i = 0, \ldots, N-1$ zu. Dieser ist eindeutig zwischen allen Entitäten mit gleichen "Geometrietyp", also Knoten, Liniensegmente, Dreiecke, Vierecke, Tetraeder, usw.

Damit können wir etwa einen Vektor mit allen Knotenpositionen wie folgt befüllen:

```
In [3]: from dune.istl import BlockVector

        indexSet = grid.indexSet
        x = BlockVector(indexSet.size(1))
        for v in grid.vertices():
            x[indexSet.index(v)] = v.geometry.center
        print("Positionen der Knoten:", x)

        y = BlockVector(indexSet.size(1))
        for e in grid.elements():
            y[indexSet.subIndex(e, 0, 1)] = e.geometry.position([0])
            y[indexSet.subIndex(e, 1, 1)] = e.geometry.position([1])
        print("Positionen der Knoten:", y)
```

```
Positionen der Knoten: ((0.000000), (0.250000), (0.500000), (0.750000), (1.000000))
Positionen der Knoten: ((0.000000), (0.250000), (0.500000), (0.750000), (1.000000))
```

1.3 Beispiel: Gebietsränder

Etwas schwieriger ist die Aufgabe, die Randpunkte zu bestimmen. DUNE kennt hier das Konzept eines Schnittes, wobei damit sowohl der Schnitt zweier Elemente als auch der Schnitt mit dem Rand gemeint ist. Gegeben ein Element können wir alle Schnitte wie folgt ablaufen:

```
In [4]: def c(e):
            return None if e is None else e.geometry.center

        for e in grid.elements():
            print("Element:", c(e), "hasBoundaryIntersections:", e.hasBoundaryIntersections())
```

```
            for i in grid.intersections(e):
                print("Intersection:", i.geometry.center, "boundary:", i.boundary, "neighbor:",
```

```
Element: (0.125000) hasBoundaryIntersections: True
Intersection: (0.000000) boundary: True neighbor: None
Intersection: (0.250000) boundary: False neighbor: (0.375000)
Element: (0.375000) hasBoundaryIntersections: False
Intersection: (0.250000) boundary: False neighbor: (0.125000)
Intersection: (0.500000) boundary: False neighbor: (0.625000)
Element: (0.625000) hasBoundaryIntersections: False
Intersection: (0.500000) boundary: False neighbor: (0.375000)
Intersection: (0.750000) boundary: False neighbor: (0.875000)
Element: (0.875000) hasBoundaryIntersections: True
Intersection: (0.750000) boundary: False neighbor: (0.625000)
Intersection: (1.000000) boundary: True neighbor: None
```

Wir können für jeden Schnitt auch die lokale Nummer der "Seitenfläche" (hier Knoten) bekommen, zu der sie gehört:

```
In [5]: for e in grid.elements():
            print("Element:", c(e), "hasBoundaryIntersections:", e.hasBoundaryIntersections())
            for i in grid.intersections(e):
                print("Intersection:", i.geometry.center, "bzw.", i.indexInInside)
```

```
Element: (0.125000) hasBoundaryIntersections: True
Intersection: (0.000000) bzw. 0
Intersection: (0.250000) bzw. 1
Element: (0.375000) hasBoundaryIntersections: False
Intersection: (0.250000) bzw. 0
Intersection: (0.500000) bzw. 1
Element: (0.625000) hasBoundaryIntersections: False
Intersection: (0.500000) bzw. 0
Intersection: (0.750000) bzw. 1
Element: (0.875000) hasBoundaryIntersections: True
Intersection: (0.750000) bzw. 0
Intersection: (1.000000) bzw. 1
```

Dies können wir etwa nutzen, um einen Vektor aufzubauen, der für jeden Knoten sagt, ob er ein Randknoten ist:

```
In [6]: m = BlockVector(grid.indexSet.size(1))
        for e in grid.elements():
            if not e.hasBoundaryIntersections():
                continue
            for i in grid.intersections(e):
                if i.boundary:
                    m[grid.indexSet.subIndex(e, i.indexInInside, 1)] = 1
        print(m)
```

```
((1.000000), (0.000000), (0.000000), (0.000000), (1.000000))
```

1.4 Beispiel: Quadraturen

Im folgenden werden wir exemplarisch die Funktion $f(x) = x^2$ auf über unserem Testgitter integrieren. Dazu verwenden wir eine Quadratur 3. Ordnung auf dem Einheitsintervall $[0, 1]$.

```
In [7]: from dune.geometry import line, quadratureRule

        f = lambda x: x[0]**2

        sum = 0
        for e in grid.elements():
            g = e.geometry
            for q in quadratureRule(line, 2):
                p = q.position
                w = q.weight * g.integrationElement(p)
                sum += w * f(p)
        print(sum)
```

```
0.333333333333
```

2 Lösung