Python und FEniCS: Verschiedenes

Python-Version: 3.6.7

FEniCS-Version: 2019.1.0

Jascha Knepper

Department Mathematik und Informatik
Universtität zu Köln

16. Mai 2019

Im Rahmen der Vorlesung *Wissenschaftliches Rechnen II* von
Prof. Dr. Axel Klawonn
SS 2019

Outline

FEniCS-Programm-Struktur

$\longrightarrow$ *Anhand eines Programmes erklären:*

- Pointer und Kopie einer Variablen
- Klassen und Objekte
- Was tut FEniCS, wenn man z.B.

```
a = inner(grad(u),grad(v))
```

schreibt? $\rightarrow$ Verkettung von Befehlen.

Parallele Ausführung

Python-Programm mit 3 MPI-Prozessen ausführen:

```
mpirun -n 3 python3 DATEI.py
```

Quellcode vergleichen

diff und kdiff3

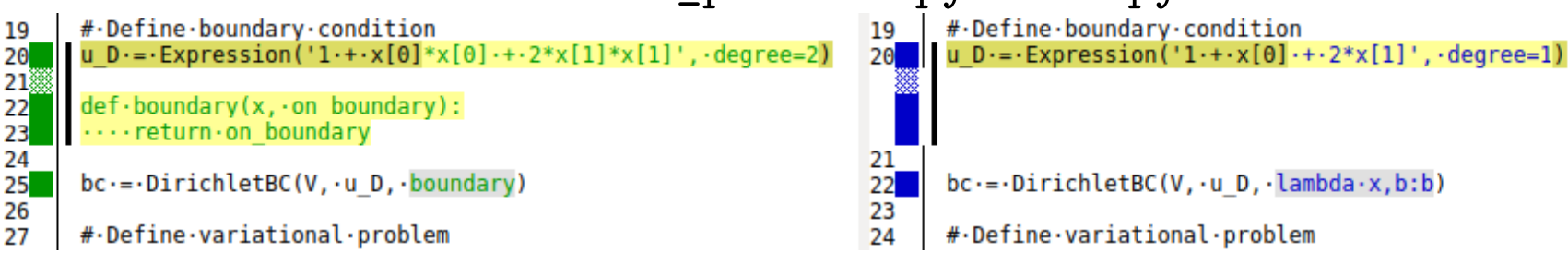
`diff` ist ein Standardprogramm (Ubuntu, MacOS).

`diff ft01_poisson.py test.py`

```
wr2@wr2:~/Documents/python/beispiele/fenics/diffusion/test$ diff ft01_poisson.py test.py
20c20
< u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
---
> u_D = Expression('1 + x[0] + 2*x[1]', degree=1)
22,25c22
< def boundary(x, on_boundary):
<     return on_boundary
<
< bc = DirichletBC(V, u_D, boundary)
---
> bc = DirichletBC(V, u_D, lambda x,b:b)
```

Unter Windows, MacOS und Ubuntu kann man z.B. `kdiff3` installieren (Ubuntu: `sudo apt-get install kdiff3`):

`kdiff3 ft01_poisson.py test.py`



Für MacOS gibt es auch das Apple-Developer-Tool `FileMerge`.

Gitter

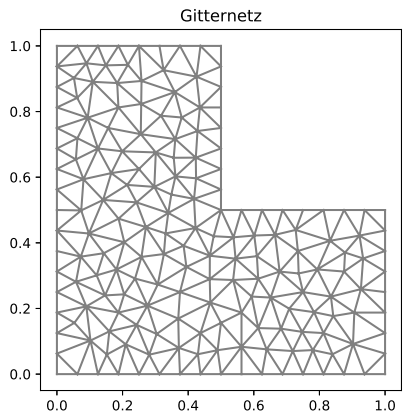
Standardgitter

Einheitsquadrat, Einheitswürfel, Rechteck, Quader:

```
mesh = UnitSquareMesh(nx,ny)
mesh = UnitCubeMesh(nx,ny,nz)
mesh = RectangleMesh(Point(ax, ay),
                     Point(bx, by),
                     nx, ny)
mesh = BoxMesh(Point(ax, ay, az),
               Point(bx, by, bz),
               nx, ny, nz)
```

Mshr: Polygon

```
import mshr as mshr
domainL_vertices = [Point(0.0, 0.0),
                    Point(1.0, 0.0),
                    Point(1.0, 0.5),
                    Point(0.5, 0.5),
                    Point(0.5, 1.0),
                    Point(0.0, 1.0)]
domain = mshr.Polygon(domainL_vertices)
mesh = mshr.generate_mesh(domain,10)
```



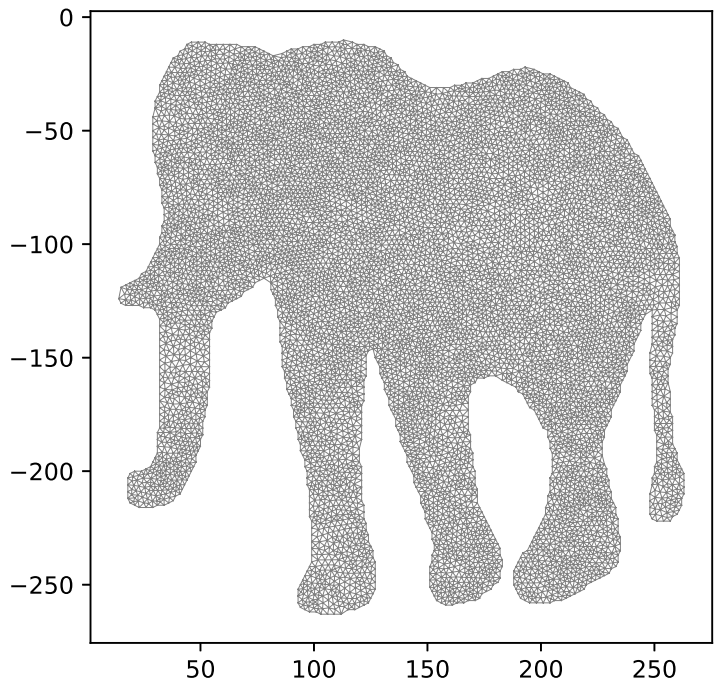
Gmsh

Gmsh-Gitter in das FEniCS-XML-Format konvertieren (Ubuntu-Terminal):

```
dolfin-convert 'gitter.msh' 'gitter.xml'
```

In FEniCS einlesen:

```
mesh = Mesh('gitter.xml')
```



FEniCS-Parameter

$\longrightarrow$ *Zeige Beispielprogramm* `fenics_parameters.py`.

Funktionen

- Standarddefinition:

```
def test(x,y):
    v = x+y
        w = x-z
        return v,w
x = 1
y = 2
_,w = test(x,y)
```

- Geschachtelte (nested) Funktionen:

```
def test(x,y):
    v = x+y
        w = x-z
        def testInner(a,b):
            return (a+v)*(b-w)
        erg = testInner(1,5)
        return erg
print(test(1,2))
```

- Lambda-Funktion:

```
test = lambda x,y: x*y
print(test(4,5))
```

Debugging im Terminal (unter Ubuntu)

Standard-Debugger

Der Standard-Debugger kann u.a. aufgerufen werden, indem man an der gewünschten Stelle im Code einen Breakpoint mit

```
import pdb; pdb.set_trace()
```

setzt. Das Programm wird wie immer mit `python3 programm.py` aufgerufen.

IPython-Debugger

Der `IPython`-Debugger ist etwas komfortabler, daher gehen wir nur auf diesen genauer ein. Unter Ubuntu kann dieser mit dem folgenden Befehl installiert werden: `sudo apt-get install python3-ipdb`. Analog zum Standard-Debugger werden Breakpoints mit

```
import ipdb; ipdb.set_trace()
```

gesetzt (der Import-Befehl ist natürlich nur einmal notwendig, kann aber auch immer dazugeschrieben werden).

```
import ipdb
x = 1
y = 2
ipdb.set_trace()
z = x*y
ipdb.set_trace()
```

Das Programm wird mit `python3 prog.py` ausgeführt; der `IPython`-Debugger wird automatisch aufgerufen. Bei Problemen kann man auch versuchen diesen direkt aufzurufen: `ipython3 prog.py`.

Debugger: Befehle

- Der Debugmodus kann mit `CTRL+D` beendet werden oder indem man `q` (**quit**) eingibt und mit Enter bestätigt.
- Markierte Zeile ausführen und zur nächsten in der aktuellen Funktion springen: `n` (**next**) eingeben (*Step* im MATLAB-Debugger).
- Markierte Zeile ausführen und zur nächsten in der aktuellen Funktion oder einer Unterfunktion springen: `s` (**step**) eingeben (*Step In* im MATLAB-Debugger).
- Bis zum `return`-Statement einer Funktion ausführen: `r` (**return**) eingeben (*Step Out* im MATLAB-Debugger).
- Codeausführung fortsetzen: `c` (**continue**).
- Lokale Variablen anzeigen: `locals()`.

Debugger: Call Stack

Ähnlich zum return-Befehl kann der *Call Stack* mit `u` (**up**) bzw. `d` (**down**) navigiert werden. Der Befehl `w` (**where**) gibt eine Übersicht über den Call Stack.

Beispiel:

Setzen wir mit `b 44` einen Breakpoint in Zeile 44, so können wir anschließend (sofern keine weiteren Breakpoints existieren) mit `c` dort hingelangen.

Mit `u` gelangt man dann an die Aufrufstelle der Funktion `test2`, d.h. in Zeile 50. Geht man eine weitere Stufe im Call Stack hoch, so springt man zur Zeile 54.

```
43 def test2(x):
44     y = x+x
45     return y
46
47 def test(x):
48     y = x*x
49     a = y
50     b = test2(a)
51     c = y
52     return y
53
54 print(test(2))
```

Mit `d` kann man wieder eine Stufe zurück. Entscheidet man sich das Programm fortzuführen, so geschieht dies an der Stelle, an die man mit `u` bzw. `d` navigiert ist.

Debugger: Befehle

- Breakpoints setzen:
  - Persistent: `b ZEILE` (**break**).
  - Temporär (gilt nur einmal): `tbreak ZEILE` (**temporary break**) (Ersatz für die Funktion *Run to Cursor* im MATLAB-Debugger).

  Achtung: In der angegebenen Zeile muss ein Befehl stehen (keine Leerzeile oder Kommentar).
  - Anstelle eine Zeile kann auch ein Funktionsname angegeben werden.
  - Man kann auch die Zeile in einer anderen Datei angeben:

    ```
    b datei.py:ZEILE
    ```

- Breakpoints auflisten (ohne `ipdb.set_trace()`): `break` eingeben.
- Breakpoints löschen (ohne `ipdb.set_trace()`): `clear` eingeben und mit `y` bestätigen.

Debugger-Start: Alternativen

- Programm direkt mit Debugger starten (ohne Breakpoints setzen zu müssen):

```
ipython3 -m ipdb prog.py
```

- Bei Fehler Debugger starten

```
from ipdb import launch_ipdb_on_exception
with launch_ipdb_on_exception():
    x = 1
    y = 2
    z = '3'
    b = x/z
    print(b)
```

Debugging: Verschiedenes

- Man sollte den Aufruf des Debuggers über `ipython3 -m ipdb prog.py` nicht mit den Breakpoints im Code der Art `ipdb.set_trace()` mischen.
- Der Debugger beherrscht Autovervollständigung (`TAB` drücken). Man kann bspw. zum Finite-Elemente-Raum aus FEniCS `V.` schreiben („`V` Punkt“) und `TAB` drücken, um die verfügbaren Funktionen und Eigenschaften anzuzeigen. Aus der gezeigten Liste wird jedoch nicht klar, ob der Befehl zu einer Variablen gehört oder einer Funktion. Bei einer Funktion muss noch eine Klammer anhängt werden: `V.dim()`.
- `type(T)` gibt den Typ der Variablen `T` an.

Konvertierung zu Numpy/SciPy-Arrays

Lösungsfunktion

Sei u eine Lösungsfunktion:

```
u = Function(V)
solve(a == F, u, bc)
```

Dann wertet der folgende Befehl u in allen Knoten aus und gibt ein Numpy-Array zurück.

```
uEval = u.vector().get_local()
```

Dieser Befehl funktioniert auch für Räume höherer Ordnung $\mathcal{P}_k$ und für `VectorFunctionSpace` und für gemischte Räume der Art

```
element = MixedElement([P2,P1,C])
V = FunctionSpace(mesh,element)
```

Eine Funktion im Unterraum erhält man z.B. über

```
(u1,u2) = u.split(True)
u1Eval = u1.vector().get_local()
```

Lösungsfunktion

Eine Alternative für $\mathcal{P}_1$-Räume ist

```
uEval = u.compute_vertex_values(mesh)
```

Für höherdimensionale und gemischte Räume muss man zunächst an die Punkteliste gelangen:

```
points = V.tabulate_dof_coordinates()
```

Wenn man die Punkteliste des i-ten Teilraumes braucht $i \in \{0, 1, \dots\}$:

```
V1 = V.sub(i).collapse()
points = V1.tabulate_dof_coordinates()
```

Anschließend kann u ausgewertet werden.

```
uEval =
    np.empty((points.shape[0],u.value_shape()[0]))
for i in range(0,points.shape[0]):
    uEval[i] = u(points[i])
```

Lösungsfunktion

Die bisherigen Konvertierungen geben Vektoren zurück. Ist u z.B. eine Lösung im Raum `V = VectorFunctionSpace(mesh,'P',2)`, so kann mit

```
uEval = u.vector().get_local()
uEval = uEval.reshape((
        u.geometric_dimension(),
        V.sub(0).dim()
        )).T
```

ein Numpy-Array der Dimension $n \times d$ erzeugt werden.

Matrix und Vektor

Wurden z.B. über

```
A, b = assemble_system(a, F, bcs)
```

oder

```
A = assemble(a)
b = assemble(F)
```

Matrix und Vektor assembliert, so kann man über

```
A = A.array()
b = b.get_local()
```

diese in dichte Numpy-Arrays konvertieren. Ist A ein Vektor, da einer der zugehörigen Finite-Elemente-Räume nur einen Freiheitsgrad hat, so wird A dennoch intern als Matrix behandelt. In diesem Fall existiert der Befehl `A.get_local()` nicht und man kann `A.array().flatten()` nutzen.

Matrix und Vektor

Es mag hilfreich sein den Befehl

```
parameters["reorder_dofs_serial"] = False
```

am Anfang des FEniCS-Programmes zu verwenden, um das Umsortieren von Freiheitsgraden zu unterbinden.

Sparse Matrix

Wurde die Bilinearform A z.B. mit `A = assemble(a)` assembliert, so kann sie anschließend mit

```
from scipy.sparse import csr_matrix
row,col,val =
    as_backend_type(A).mat().getValuesCSR()
A = csr_matrix((val,col,row),
    shape=[A.size(0),A.size(1)])
```

in das SciPy-unterstützte CSR-Format konvertiert werden. Obige Methode setzt voraus, dass `PETSc` als Lineare-Algebra-Backend verwendet wird (default). Soll `Eigen` verwendet werden, siehe nächste Folie.

Sparse Matrix - Eigen-Backend

Am Anfang des Programms nutzt man folgenden Befehl:

```
parameters['linear_algebra_backend'] = 'Eigen'
```

Konvertierung in das SciPy-unterstützte CSR-Format:

```
from scipy.sparse import csr_matrix
row,col,val = as_backend_type(A).data()
A = csr_matrix((val,col,row),
    shape=[A.size(0),A.size(1)])
```

Sparse Matrix

Mit `numpy.array(A.todense())` kann die Matrix in das dichte Numpy-Format konvertiert werden und mit `A.tocsc()` in das Sparse-Format CSC.
Achtung: Numpy-Arrays nutzen das Skalarprodukt für die Multiplikation, SciPy-Sparse-Matrizen das Sternchen.

```
Adense = numpy.array(A.todense())
print(numpy.linalg.norm(numpy.dot(Adense,u)-b))
print(numpy.linalg.norm(A*u-b))
```

Export zu MATLAB

Export

Hat man, wie auf Folie 19, Numpy-Arrays A und b erzeugt, so kann man diese in eine MAT-Datei exportieren:

```
import scipy.io
scipy.io.savemat('data.mat',
    mdict={'A':A,'b':b})
```

Auch Sparse-Matrizen, siehe Folie 21, werden automatisch in das entsprechende MATLAB-Format konvertiert und (dünnbesetzt) gespeichert.

Für nützliche Hinweise bzgl. Knotensortierung etc., siehe die Folien 25, 26, 27 und 33.

Sortierung der Freiheitsgrade: Reordering

Die Sortierung der Freiheitsgrade stimmt i.d.R. nicht mit den Gitterpunkten überein. Global lässt sich dieses Verhalten ausschalten:

```
parameters["reorder_dofs_serial"] = False
```

Sortierung der Freiheitsgrade: Gemischte Räume und Räume höherer Ordnung

Angenommen wir haben einen gemischen Raum der Art

```
P2 = VectorElement('P',triangle,2)
P1 = FiniteElement('P',triangle,1)
element = MixedElement([P2,P1])
V = FunctionSpace(mesh,element)
```

und $\mathcal{V}_{P1}$ sind die $\mathcal{P}_1$-Knoten und $\mathcal{V}_{P2}$ die $\mathcal{P}_2$-Knoten, dann sind die Freiheitsgrade wie folgt sortiert:

$$(\underbrace{\mathcal{V}_{P1}, \mathcal{V}_{P2}}_{dof(u_x)}, \underbrace{\mathcal{V}_{P1}, \mathcal{V}_{P2}}_{dof(u_y)}, \underbrace{\mathcal{V}_{P1}}_{dof(p)})$$

Dies bedeutet, dass an erster Stelle alle Freiheitsgrade des Raumes `FunctionSpace(P2)` stehen und anschließend die des Raumes `FunctionSpace(P1)`. Die Freiheitsgrade des $\mathcal{P}_2$-Raumes sind zunächst bzgl. der x-Koordinate und dann bzgl. der y-Koordinate sortiert. Darin stehen dann jeweils zuerst die $\mathcal{P}_1$-Knoten und anschließend die $\mathcal{P}_2$-Knoten.

Sortierung der Freiheitsgrade: Punktelisten

$\mathcal{P}_1$-Punkte und Dreiecke:

```
x = mesh.coordinates()
tri = mesh.cells()
```

Punktelisten für gemischte und höherdimensionale Räume:

```
points = V.tabulate_dof_coordinates()
V1 = V.sub(0).collapse()
points_V1 = V1.tabulate_dof_coordinates()
```

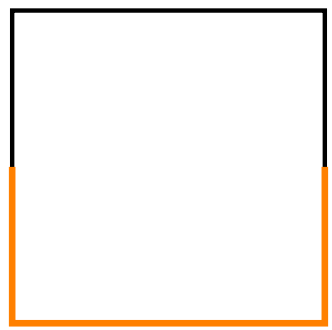
Bemerkung: Ggfs. ist auch der Befehl `V.dofmap().dofs()` hilfreich.

Randbedingungen

Standard-Dirichlet-Randbedingung

Für die Definition von Dirichletbedingungen wird der Befehl `DirichletBC` genutzt.

```
bc = DirichletBC(V,Constant((0,0)),lambda
    x,on_boundary: on_boundary and (abs(x[1]) <=
    0.5))
```



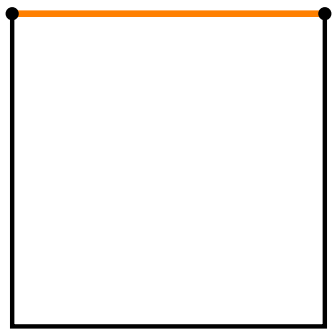
Wichtig: Man kann damit auch Dirichletbedingungen im Innern und nicht nur auf dem Rand setzen. Dann darf man natürlich nicht den Parameter `on_boundary` nutzen.

Punktweise Dirichlet-Randbedingung

„Offene“ Randbedingung:

$$u(x, y) = (1, 0) \text{ auf } \{(x, y) : y = 1, x \in (0, 1)\}.$$

```
def top(x, on_boundary): return (x[1] >
    1.0-DOLFIN_EPS) and (x[0] > DOLFIN_EPS) and
    (x[0] < 1-DOLFIN_EPS)
bc = DirichletBC(W.sub(0), Constant((1,0)),
    top, method='pointwise')
```



Achtung: Man darf hierbei nicht mehr den Parameter `on_boundary` verwenden, da dieser immer `False` ist.

Punktweise Dirichlet-Randbedingung

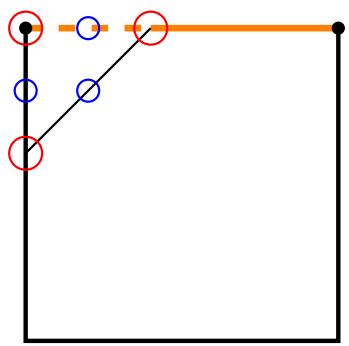
Würde man für die Randbedingung

$$u(x, y) = (1, 0) \text{ auf } \{(x, y) : y = 1, x \in (0, 1)\}.$$

weiterhin

```
bc = DirichletBC(W.sub(0), Constant((1,0)), top)
```

nutzen, so werden bei $\mathcal{P}_2$-Elementen u.U. $\mathcal{P}_2$-Knoten nicht richtig berücksichtigt:



Die $\mathcal{P}_1$-Knoten sind hier rot markiert und die $\mathcal{P}_2$-Knoten blau. Der $\mathcal{P}_2$-Knoten auf der oberen Kante müsste die Bedingung $u = (1, 0)$ erfüllen. Dies ist jedoch nicht der Fall.

Assemblierung

Systemmatrix- und Vektor assemblieren und Randbedingungen **symmetrisch** eliminieren:

```
A, b = assemble_system(a, F, bcs)
```

Systemmatrix- und Vektor assemblieren und **keine Randbedingungen eliminieren**:

```
A = assemble(a); b = assemble(F)
```

Nachträgliche **nicht-symmetrische** Eliminierung der Randbedingungen:

```
bc.apply(A); bc.apply(b)
```

bzw.

```
bc.apply(A,b)
```

Ist `bcs` eine Liste aus Randbedingungen, so muss über eine Schleife jede einzelne Randbedingung angewandt werden.

Assemblierung: Symmetrie der RB

- Symmetrische Eliminierung:

$$\begin{pmatrix} A_{II} & 0 \\ 0 & D \end{pmatrix} \cdot \begin{pmatrix} u_I \\ u_D \end{pmatrix} = \begin{pmatrix} b_I - A_{ID} u_D \\ D u_D \end{pmatrix}$$

- Nicht-symmetrische Eliminierung:

$$\begin{pmatrix} A_{II} & A_{ID} \\ 0 & D \end{pmatrix} \cdot \begin{pmatrix} u_I \\ u_D \end{pmatrix} = \begin{pmatrix} b_I \\ D u_D \end{pmatrix}$$

D ist eine invertierbare Diagonalmatrix, z.B. $D = I$ (Einheitsmatrix), siehe Folie 33.

Assemblierung: Verschiedenes

Steifigkeitsmatrix und Lastvektor können mit eliminierten Randbedingungen (siehe Folie 31) z.B. in MATLAB weiterverwendet werden (siehe Folie 24). Abgesehen von der Knotensortierung (siehe 25 und folgende Folien) sei angemerkt:

Die Diagonaleinträge, die zu Dirichletknoten gehören, stehen nicht unbedingt Einsen (vgl. mit Folie 32). Im Allgemeinen kann mit einer invertierbaren Diagonalmatrix skaliert werden.

Meine Beobachtungen: Wurden die Randwerte nicht-symmetrisch eliminiert, so ist D die Einheitsmatrix. Bei symmetrischer Eliminierung ist der Eintrag D_{ii}, der zum Knoten x_i gehört, die Elementmultiplizität des Knotens, d.h. die Anzahl an Elementen, die den Knoten x_i enthalten. Für einen $\mathcal{P}_2$-Knoten auf dem Rand ist dies z.B. $D_{ii} = 1$ und für einen $\mathcal{P}_2$-Knoten im Innern ist $D_{ii} = 2$.

Löser außerhalb von FEniCS (SciPy, Numpy)

$\longrightarrow$ *Zeige Programme.*