1 Visualisierung des Tapes

Die Fragestellung dieses Abschnittes ist: Wie lässt man sich das Tape, das Dolfin adjoint erstellt, anzeigen? Der Befehl, den man dafür verwendet lautet: `get_working_tape()` um die entsprechende Instanz zu erzeugen. Wichtig dabei ist, dass man den Befehl im Code erst an einer Stelle einfügt nachdem alle relevanten Rechnungen (die man später visualisiert haben möchte) ausgeführt wurden. In unserem Fall heißt das also den Aufruf erst nach der Berechnung von `Jhat` einzufügen, genauso wie in Abbildung 1. Wir können jetzt auf der Instanz des Tapes den Befehl `visualise` ausführen, dieser Befehl kann mehrere Parameter erhalten, für unseren Fall reicht es den Zielordner anzugeben (hier heißt der Zielordner: tape). Damit erklärt sich die Eingabe von `get_working_tape().visualise("tape")`, wie in Abbildung 1. Wir benötigen nichts weiteres aus der Instanz des Tapes, also beenden wir das Programm mit `exit(0)`. Damit die Visualisierung funktioniert muss außerdem TensorFlow in Python installiert sein.

```
# Define reduced functional and test gradients
Jhat = ReducedFunctional (assemble (J), Control (v), derivative_cb_post = callback)

get_working_tape().visualise ("tape")
exit(0)
```

Abbildung 1: Positionierung von get_working_tape() im Code

Führt man das Programm um die so ergänzten Zeilen aus, so erscheinen im Terminal zusätzliche Ausgaben wie in Abbildung 2:

```
Run the command line:
--> tensorboard --logdir=tape
Then open http://localhost:6006/ in your web browser.
```

Abbildung 2: Zusätzliche Ausgaben

Folgt man diesen Anweisungen, also zuerst das Ausführen von `tensorboard --logdir=tape` im Terminal und dann das Öffnen des angegeben Links in einem Browser, so gelangt man zur Darstellung des Codes in einer Art Baumdiagramm, für unser Beispiel ist der Baum in Abbildung 3 zu sehen.

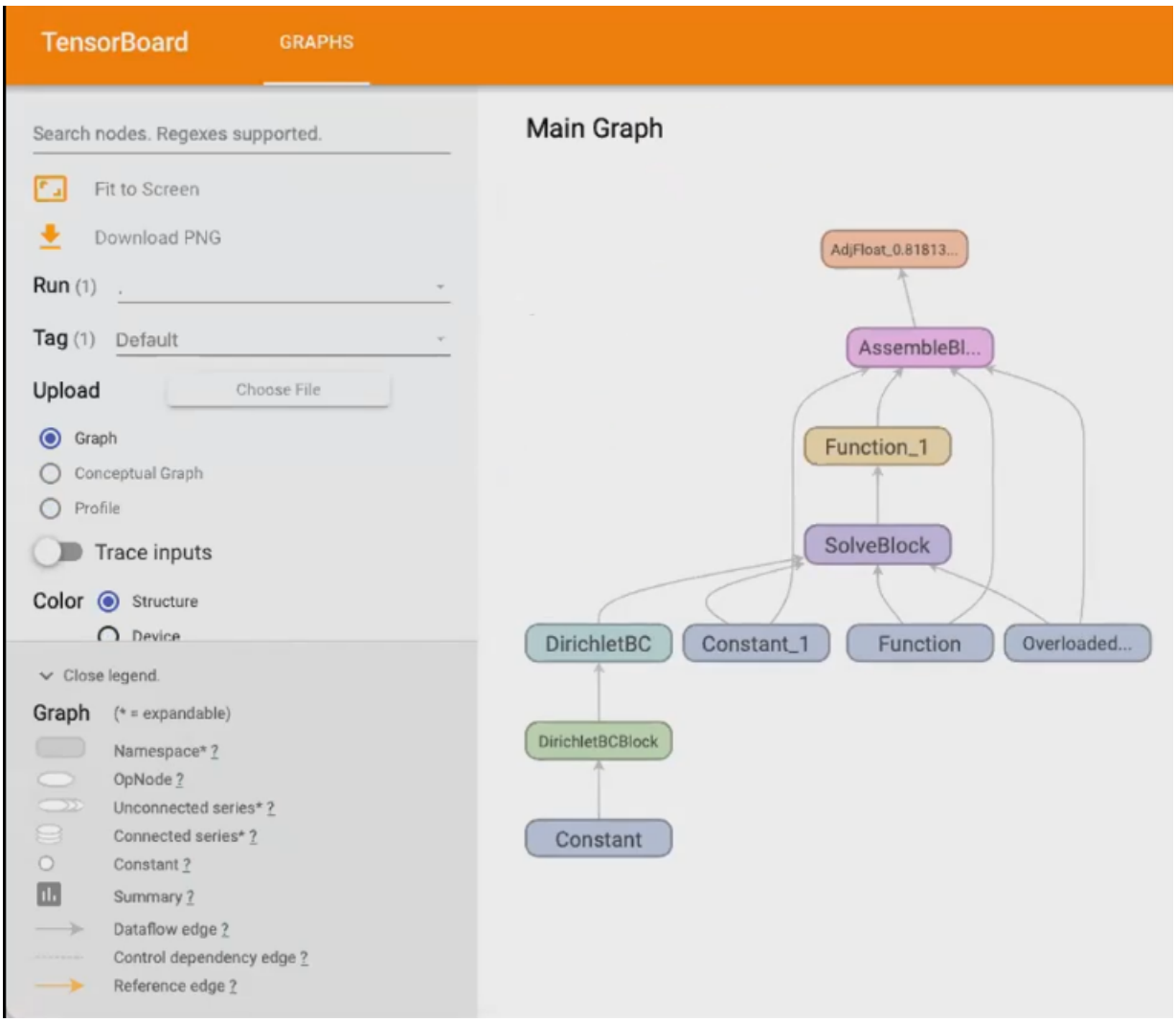

Abbildung 3: Baumdiagramm unseres Algorithmuses

In diesem Diagramm kann man jetzt durch Doppelklicken auf die verschiedenen Knoten „Untermenüs“ öffnen, dort stehen zusätzliche Informationen, beispielsweise Namen von Funktionen, von Konstanten, ... (sofern man natürlich entsprechende Namen im Code vergeben hat). Außerdem kann das Diagramm übersichtlicher gestalten werden, indem uninteressante Knoten aus dem Hauptgraphen, mittels Rechtsklick und Auswählen der Option „remove from main graph“, entfernt werden.

Eine weitere Möglichkeit Übersicht im Graphen zu erhalten ist, bereits das Tape in kleinere logische Blöcke zu zerlegen. Dies wird mit der Codezeile
`with get_working_tape().name_scope{...}` erreicht (dem Befehl wird ein Name in geschweiften Klammern übergeben). Anschaulich gesprochen schreibt das Programm alles was in der selben Zeile hinter `with` (oder eingerückt unterhalb) von `with` steht in einen eigenen Knoten und versteckt quasi Neben- oder Unterrechnungen. Der Graph für unser Beispiel sieht dann aus wie in Abbildung 4.

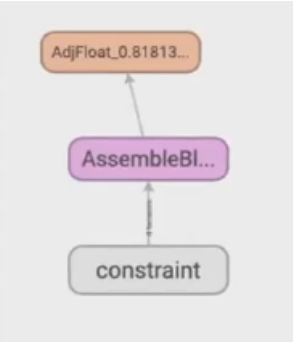

Abbildung 4: Baumdiagramm mit verborgener Rechnung im Knoten constraint

Klappt man jetzt den Knoten „constraint“ auf so erscheint wieder der vollständige Baum.

2 Adjoint ohne dolfin-adjoint

Die Frage, die in diesem Abschnitt beantwortet werden soll ist: Wie können wir unseren Algorithmus schreiben, ohne dolfin-adjoint zu verwenden?

Das mathematische Problem welches dieser Fragestellung zu Grunde liegt ist jenes aus der ersten Vorlesung unseres Praktikums. Kurz noch einmal zur Wiederholung: Wir wollen das Funktional

$$J(u, v)$$

bezüglich v minimieren. Dabei ist das u, welches zu v gehört gegeben als der Minimierer von

$$E(u, v)$$

In unserer Anwendung sind die Variablen so zu verstehen:

Name	Bedeutung
u	elastische Verschiebung
v	Phasenfeld
J	Zielfunktional
E	elastische Energie

Wenn wir einen eindeutigen Minimierer von $E(u, v)$ für ein v finden so hängt dieser nur von v ab und wir bezeichnen diesen mit $u(v)$, dann können wir vereinfacht schreiben:

$$\widehat{J}(v) := J(u(v), v).$$

Eine Anmerkung zur Notation: der Ausdruck $\widehat{J}_{,v}(v)$ bezeichnet die Ableitung von $\widehat{J}$ nach v in eine beliebige Richtung. Der Ausdruck $E_{,uv}(u, v)p$ hingegen bezeichnet die zweimalige Ableitung von E, einmal nach u in die Richtung p und einmal nach v (wieder in beliebige Richtung).

In vergangenen Vorlesungen haben wir bereits gesehen, dass gilt:

$$\widehat{J}_{,v}(v) = E_{,uv}(u,v)p + J_{,v}(u,v) \tag{1}$$

(es ist zu beachten, dass die beliebige Ableitungsrichtung in $\widehat{J}_{,v}$ und in $J_{,v}$ die selbe ist!)

Wir kennen u und p aus der Gleichung 1 nicht. Diese müssen wir wie folgt berechnen:

u ist definiert durch:	$E_{,u}(u,v) = 0$
p ist definiert durch:	$E_{,uu}(u,v)p = -J_{,u}(u,v) (\rightarrow$ linear in $p)$

Natürlich müssen wir alles in vernünftiger Reihenfolge berechnen, unser Problem folgt also diesem mathematischen Ablauf:

1. Berechne u aus $E_{,u}(u,v) = 0$
2. berechne p aus $E_{,uu}(u,v)p = -J_{,u}(u,v)$
3. berechne $\widehat{J}_{,v}$ aus $\widehat{J}_{,v}(v) = E_{,uv}(u,v)p + J_{,v}(u,v)$

Wir können diesen Ablauf in FENICS so formulieren:

1. ```
   duE = derivative(E,u)
   solve(duE==0,...)
   ```
2. ```
   duduE = derivative(duE, u)
   duJ = derivative(J,u)
   p = TrialFunction(...)
   solve(action(duduE, p) == -duJ,...)
   ```
3. ```
   dvJhat = derivative(action(duE,p),v) + derivative(J,v)
   g = Function(...)
   assemble(dvJhat, tensor = g.vector())
   ```

2.1 lineares Gleichungssystem statt Newton-Verfahren

Wir sehen die Implementierung ist in ihrer Formulierung sehr nahe an der mathematischen Formulierung, nichtsdestotrotz ist dieser Ablauf genau das was dolfin-adjoint auch berechnet, wir wollen ja aber schneller werden mit unserem Algorithmus. Wir erreichen das, dadurch, dass wir unser Problem mathematisch weiter vereinfachen (was dolfin-adjoint nicht kann). Wir beobachten zuerst:

$$E = \underbrace{\text{stored}}_{\text{quadratisch in } u} - \underbrace{\text{compliance}}_{\text{linear in } u} \tag{2}$$

Wir können das ausnutzen, d.h. $E_{,u}$ ist affin in u (beim Ableiten wird der u^2-Term zu einem u-Term und der u-Term wird konstant), also ist $E_{,u}$ ein lineares Gleichungssystem. In FENICS formuliert sich das so:
   ```

```
ut = TrialFunction()
duE2 = replace(duE, {u:ut})
solve(lhs(duE2)==rhs(duE2),...)
```

Dieses Umschreiben bringt uns den Vorteil, dass wir statt einem Newton-Löser wie im Ablauf unter Punkt 1. zu sehen, nun nur ein lineares Gleichungssystem lösen müssen.

2.2 Ersetzen von lhs(...) und rhs(...)

Um das noch weiter zu verbessern betrachten wir genau was in `lhs(duE2)` steht: Das ist definiert als die erste Ableitung von E, wobei darin der Ausdruck u durch den Ausdruck ut ersetzt wurde. Weiterhin bleiben, durch den Befehl `lhs(...)`, in `lhs(duE2)` nur die Teile stehen, in denen ut vorkommen. Wir behaupten jetzt das gilt:

```
lhs(duE2) = action(duduE,ut)
```

Das erklärt sich, wenn man sich überlegt was in `action(duduE,ut)` passiert: Wir leiten E zweimal ab und setzt in für eine Ableitungsrichtung ut ein. Anhand eines Beispiels wird deutlicher was passiert:

Beispiel. *Sei* $E(u) = au^2+bu+c \implies duE(u) = 2au+b$ *wir rechnen beide Möglichkeiten durch:*

`lhs(duE2)`	`action(duduE,ut)`
ersetze in duE u durch $ut \to 2aut + b$	leite duE nach u ab $\to duduE = 2a$
verwende nur Terme mit $ut \to 2aut$	setzte ut als Ableitungsrichtung$\to 2aut$
$\implies$ `lhs(duE2) = action(duduE,ut)`	

Insgesamt gilt diese Gleichheit da `duE2` linear in u ist.

Mit einer ähnlichen Argumentation lässt sich auch `rhs(duE2)` umformulieren: Die rechte Seite ist der Konstante Term von $duE2$ und wir wissen, dieser Term kommt aus der Compliance, genauer: es ist die Ableitung der Compliance. Wir erhalten also:

```
rhs(duE2)=-(-derivative(Comp,u))=derivative(Comp,u)
```

(ein Minuszeichen kommt aus der Gleichung 2 und das andere aus der Äquivalenzumformung die in `rhs(...)` passiert.). Wir können also unseren schon modifizierten 1. Schritt weiter umformulieren, indem man schreibt:

```
duC = derivative(Comp,u)
solve(action(duduE,ut)==duC,...)
```

Was haben wir dadurch erreicht? Auf den ersten Blick haben wir unsere Gleichungen nur umgeformt, was uns natürlich nichts an Rechenleistung erspart, dennoch ist etwas sehr nützliches passiert: Betrachten wir unseren Ablauf aus dem ursprünglichen FENICS-Code, dort steht unter Punkt 2.: `solve(action(duduE,p)==-duJ,...)`, d.h. Unsere Gleichungssysteme in Schritt 1. (nach dem Umformulieren) und in Schritt 2. sind zumindest

schon auf der linken Seite identisch (beidesmal steht dort: `action(duduE,p)`). Betrachten wir die rechte Seite noch etwas genauer, dort steht `-duJ`. Wir wissen bereits wie sich J zusammensetzt:

$$J = \underbrace{\text{phaseField}}_{\text{abhängig von } v} + \underbrace{\text{compliance}}_{\text{abhängig von } u}$$

Wenn wir jetzt aber J nach u ableiten fällt der Phasenfeldterm komplett weg, da dieser ja nur von v abhängt, es gilt also:

```
duJ = duC
```

Tatsächlich sind also auch die rechten Seiten der zwei Gleichungen sehr ähnlich, sie unterscheiden sich nur um den Faktor -1. An diesem Punkt können wir uns nun aber Rechenarbeit sparen, man muss natürlich nicht zweimal das (fast) gleiche Gleichungssystem lösen, da bis auf ein -1 in Schritt 2 alles gleich bleibt, sparen wir uns das rechnen und geben als Lösung des Gleichungssystems nur -1-mal die Lösung aus Schritt 1 an. Wir ersetzten also Schritt 2 durch

```
p.assign(-ut)
```

Wenn wir jetzt unsere Erkenntnisse zusammenfassen erhalten wir:

1. ```
   duE = derivative(E,u), duduE, duC
   solve(action(duduE,ut)==duC,...)
   ```

2. ```
   p.assign(-ut)
   ```

3. ```
   dvJhat = derivative(action(duE,p),v) + derivative(J,v)
   g = Function(...)
   assemble(dvJhat, tensor = g.vector())
   ```

Wir sehen: in Schritt 2. fehlt nun das Gleichungssystem, wir sparen uns somit also wirklich Rechenleistung.

2.3 scipy-minimize

Wenn wir uns dolfin-adjoint sparen wollen, dann dürfen wir nicht das minimize aus dem dolfin-adjoint Paket verwenden, sondern müssen mit dem scipy-minimize arbeiten. Das kann aber nicht direkt auf FENICS-Funktionen arbeiten, wir müssen also einiges umschreiben. Schauen wir zuerst was wir dem scipy-minimize übergeben müssen:

```
scipy.minimize(Jhat, arr_v, jac=DJhat, ...)
```

- Das Argument in dem optimiert werden soll, kann keine FENICS-Funktion sein. Das liegt daran, dass der `scipy.minimize(...)` Befehl im $\mathbb{R}^n$ Vektorraum arbeitet und FENICS-Funktionen nicht automatisch in diesen Vektorraum übertragen werden. $\Longrightarrow$ Wir müssen also aus unserer FENICS-Funktion `v` in ein numpy-Vektor `arr_v` transformieren.
```

- `Jhat` soll aber weiterhin eine FENICS-Funktion sein, nur wollen wir dieser jetzt `arr_v` übergeben, also müssen wir am Beginn der Definition von `Jhat arr_v` in eine FENICS-Funktion umwandeln und am Ende einen Float zurückgeben.
- Gleiches gilt auch für `DJhat`: Auch hier übergeben wir der Funktion `arr_v`, welches am Beginn wieder in eine FENICS-Funktion umgewandelt werden muss. Am Ende von `Jhat` muss die FENICS-Funktion `g` entsprechend wieder in eine numpy-Version umgewandelt werden.

Der Code um eine FENICS-Funktion in ein numpy-Array umzurechnen sieht wie folgt aus:

```
arr_v = v.vector().get_local()
```

Und um aus einem numpy-Array eine FENICS-Funktion zu machen:

```
v.vector().set_local(arr_v)
```

Außerdem müssen wir beachten, dass die Grenzen ebenfalls numpy-Arrays sein müssen, diese waren bisher ebenfalls FENICS-Funktionen. Außerdem müssen wir einmal transponieren, wir schreiben also:

```
bounds = transpose([arr_low_bound, arr_up_bound])
```

2.4 Parallelisierung mit MPI

Die Parallelisierung läuft in etwa so ab: In jedem Prozess läuft das selbe Python-Programm. Wird das Mesh aufgerufen, so partitioniert FENICS dieses und jeder Prozess erhält nur einen Teil der Freiheitsgrade. Das bedeutet jedoch, dass die Prozesse untereinander kommunizieren müssen und zwar genau an den Rändern der Aufteilung des Meshes. Beispielsweise müssen in jeder Iteration beim Lösen eines Gleichungssystems die Freiheitsgrade am Rand synchronisiert werden. Im Normalfall geschieht das alles von FENICS automatisiert. Wir müssen jetzt aber an einer Stelle aufpassen, das `scipy.minimize(...)` arbeitet nicht automatisch parallel! Wir müssen also sicherstellen, dass jeder `scipy.minimize(...)` Aufruf synchron mit allen diesen Aufrufen läuft, im Endeffekt mach zwar jeder dieser Aufrufe das selbe, das ist aber letztendlich der Art der Parallelisierung geschuldet. Wir parallelisieren quasi nur den FENICS-Teil des Codes und lassen den scipy-Teil auf jedem Prozess laufen. Wir müssen also an der Stelle der Umwandlung von FENICS zu scipy und Umgekehrt etwas modifizieren.

Zuerst den Fall von einer FENICS-Funktion zu einem numpy-Array. Hier benötigen wir zu Beginn eine Liste aller globalen Indizes, diese erhalten wir durch:

```
global_range = numpy.arange(V.dim(),dtype='I')
```

In jedem Prozess muss dann folgendes ausgeführt werden, um den Koeffizientenvektor zusammen zu bauen:

```
arr = fun.vector().gather(global_range)
```

Betrachten wir nun die andere Richtung. Hier benötigen wir die lokalen Indizes:

```
local_begin, local_end = fun.vector().local_range()
```

Und jetzt setzen wir noch die Funktion aus den lokalen Indizes zusammen:

```
fun.vector().set_local(arr[local_begin:local_end])
fun.vector().apply('insert')
```

Dabei sorgt die letzte Zeile dafür, dass FENICS über veränderte Vektorwerte informiert wird.