EBFieldSolver Dokumentation

Die Skripte im `EBFieldSolver` sind entwickelt als Unterstützung zur einfachen Erstellung eigener Darstellungen von statischen und dynamischen Feldern. Mit ihnen können Feldgleichungen numerisch mit dem Nabla-Operator oder direkt im zwei- oder dreidimensionalen Raum gelöst und betrachtet werden. Dazu stehen einige Skripte, gelistet in in Tab. 1, zur Verfügung. Dabei sind alle relevanten Unterschiede aufgeführt. Zum Einstieg wird die Lösung und Darstellung von statischen Feldern für den zweidimensionalen Fall empfohlen. Der Schwierigkeitsgrad wird durch die Darstellung für den dreidimensionalen bzw. dynamischen Fall gesteigert.

Skript	Aufgabe
`FieldObject.py`	Definition von Objekten, die ein elek. oder magnetisches Feld erzeugen; Berechnung des Feldes bzw. Potentials an einem Punkt im Raum aus Objekteigenschaften
`FieldOperator.py`	Bereitstellung von Hilfsfunktion zur numerischen Berechnung des Gradienten, Divergenz und Rotation einer Funktion an einem Punkt im Raum
`FieldUtil.py`	Diverse Hilfsfunktionen: Berechnung des Gesamtfeldes für alle Koordinaten in einem Intervall; Skalierung von Vektoren etc.
`FieldAnimation.py`	Initialisierung von Achsen mit Beschriftung, Grenzen, Größen etc.; Rendern der Plots als .png oder zusammengefasst als .gif
`StaticField.py`	Instanziierung von statischen felderzeugenden Objekten und Raumkoordinaten; Durchführung der Berechnungen zum Plotten der Feldlinien, Potentiallinien etc.
`DynamicField.py`	Instanziierung von dynamischen felderzeugenden Objekten, Raumkoordinaten und Zeitfenster; Durchführung der Berechnungen zum Plotten der Feldlinien, Potentiallinien, Vektorfelder etc.

Tabelle 1: Übersicht der Skripte und dessen Aufgabe

`FieldObject.py` umfasst alle Klassen zu felderzeugenden Objekten. Dabei werden Objekte durch ihre Position, Stärkeparameter (z.B. Ladung, Stromstärke) und Ausdehnung definiert. Weiter definieren die Objekte eine konkrete Form, elektrische Feldstärke oder megnetische Flussdichte und evtl. ein (elektrisches oder Vektor-) Potential.

`StaticField.py` enthält die Deklarationen der Koordinaten mit einer feinen Gitterstruktur auch Mesh genannt. Aus den Koordinaten und den instanziierten felderzeugenden Objekten kann das statische Gesamtfeld mit Hilfe der Funktionen aus den Skripten `FieldOperator.py` und `FieldUtil.py` berechnet werden. Die Darstellung erfolgt dann mit den Funktionen `quiver()` für Vektorfelder und `streamplot()` für Stromlinien aus `matplotlib.pyplot`. Im Anschluss werden die Achsen definiert inkl. der Beschriftung, Grenzen, Größen und als .png mit dem Skript `FieldAnimation.py` gespeichert.

`DynamicField.py` ist praktisch analog zum statischen Fall aufgebaut. Zusätzlich kann ein Objekt instanziiert werden, welches ein zeitlich verändertes Feld erzeugt. Dazu muss neben den räumlichen Koordinaten noch eine Menge von zeitlichen Koordinaten übergeben werden. Zu jedem Zeitpunkt wird dann das Feld berechnet und als .png gespeichert. Im Anschluss werden alle Feldbilder zu einer Animation im .gif Format zusammengefasst. Die Funktion dazu liegt ebenfalls im Skript `FieldAnimation.py`.

Anleitung zur Erstellung eines Feldbildes

1 Definition des felderzeugenden Objekts

Damit ein Feldlinienbild geplottet werden kann, braucht es zunächst einmal ein Objekt welches ein solches Feld überhaupt erzeugen kann. Beispielsweise kann man dazu die magnetische Flussdichte um einen langen dünnen Draht berechnen, der eine konstante Stromdichte $\vec{j} = j_0\vec{e}_z$ führt. Die Stromdichte zeigt parallel zum Draht in die Bildschirmebene (z-Achse), sodass man den Querschnitt (x-y-Ebene) des Drahtes betrachtet. Das Vektorpotential eines solchen Drahtes kann ausgedrückt werden mit

$$\vec{A}(\vec{r}) = \begin{bmatrix} 0 \\ 0 \\ \frac{-\mu_0 I}{4\pi} \ln\left(\vec{r} - \vec{r}_0\right) \end{bmatrix}$$

mit der magnetischen Feldkonstante μ_0, der elektrischen Stromstärke I, der Position des Drahtmittelpunktes $\vec{r}_0$ und dem Betrachtungspunkt $\vec{r}$ wobei $r^2 = x^2 + y^2$ gilt.

Der Draht wird folglich als Objektklasse definiert und der Konstruktor erhält die elektrische Stromstärke des Drahtes und die Position des Drahtmittelpunktes. Es folgt die Klasse Leiter

```
class Conductor:
    mu_0 = 4 * np.pi * 10E-7
    const = (mu_0 / (2 * np.pi))

    def __init__(self, I, r0):
        self.I = I
        self.r0 = r0

    def A_field(self, x, y, z):
        mag = -(1. / 2.) * self.const * self.I
        Ax = 0.
        Ay = 0.
        Az = mag * np.log((x - self.r0[0])**2 + (y - self.r0[1])**2)

        return [Ax, Ay, Az]
```

Die Funktion `A_field()` liefert die korrespondierenden x-, y- und z-Komponenten des Vektorpotentials in kartesischen Koordinaten zurück, sodass diese als Höhenlinien geplottet oder weiter verwendet werden können. Es können im Folgenden beliebig viele Leiter in einer Liste gespeichert werden. In der späteren Berechnung des Gesamtfeldes wird über alle übergebenen Objekte in dieser Liste iteriert und das Gesamtfeld folgt dann aus der Superposition aller Teilfelder.

2 Berechnung des Feldes im Skript

Das Vektorpotential ist gerade so definiert, dass $\vec{B}(\vec{r}) := \nabla \times \vec{A}(\vec{r})$ gilt. Das bedeutet, dass die magnetische Flussdichte durch die Rotation des Feldes $\vec{A}(\vec{r})$ berechnet werden kann. Im ersten Schritt müssen dazu die Betrachtungskoordinaten als Mesh initialisiert werden. Die z-Achse wird mit Nullen aufgefüllt und praktisch ignoriert, da nur der Querschnitt des Leiters betrachtet wird. Über die Funktion `total_diff()` aus `FieldUtil.py` wird die Rotation mit dem Parameter `nabla='curl'` auf das Vektorpotential `f='A_field'` aus der zuvor definierten Klasse ausgeführt.

```
xx, yy = np.meshgrid(np.linspace(-xy_max, xy_max, n_xy),
                     np.linspace(-xy_max, xy_max, n_xy))
zz = np.zeros((len(xx), len(xx)))
.
.
.
rot_Ax, rot_Ay, rot_Az = util.total_diff(xx, yy, zz,
                                         conductors,
                                         f='A_field',
                                         nabla='curl')
rot_A_norm = np.hypot(rot_Ax, rot_Ay)
```

Die z-Komponente des B-Feldes ist zur Darstellung irrelevant und für die farbliche Kennzeichnung der Flussdichte, ist es praktisch den Betrag der Flussdichte mit der Funktion `hypot()` zu berechnen.

3 Plotten des Feldes im Skript

Um das Feld zu plotten, stehen grundsätzlich mehrere Optionen zur Verfügung. Eine geeignete Variante für einfache Felder ist z.B. die Funktion `streamplot()`, die zwei 2D-Mesh und zwei 2D Listen mit den jeweiligen x- und y-Beträgen der Feldvektoren übergeben bekommt.

```
Ax, Ay, Az = util.total_field(xx, yy, zz,
                              conductors,
                              f='A_field')
A_levels = np.linspace(np.min(Az), np.max(Az), 7)
.
.
.
plt.streamplot(xx, yy,
               rot_Ax, rot_Ay,
               color=np.log(rot_A_norm), cmap='cool')
plt.contour(xx, yy, Az, A_levels)
```

Das Ergebnis sind feine Linien, die sich entlang der Feldlinien der Flussdichte erstrecken und dann abgetrennt erscheinen, wenn bereits eine andere Stromlinie am zugehörigen Ankerpunkt des Mesh liegt. Die Feldstärke wird durch einen farblichen Verlauf von magenta bis cyan kenntlich gemacht durch Übergeben einer Colormap `cmap='cool'`. Zusätzlich werden die Höhenlinien des Vektorpotentials berechnet mit `total_field()` und als entsprechende Umrandungen mit `contour()` geplottet. Die Stufen der Höhenlinien müssen dabei, u.U. wie hier in sieben Stufen vom Minimum bis zum Maximum der z-Komponente, manuell berechnet werden.

4 Speichern als .png-Datei

Standardmäßig werden durch Aufrufen der Funktion `window` die Achsen mit x und y beschriftet und eine feste Größe bei gleichem Seitenverhältnis festgelegt. Alternativ können diese Parameter inkl. des zu betrachteten Wertebereichs angegeben werden.

```
anim.window()
anim.render_frame(loc='conductors')
```

Die Funktion `render_frame` erzeugt dann den Plot als .png und speichert diesen unter dem Namen `loc='NAME'` mit einer fortlaufenden Nummer im gleichen Verzeichnis wie das ausführende Skript.

5 Dynamische Felder

Erweiterung der Objekte um eine Klasse mit einem zeit- und ortsabhängigen Feld.

```
class HertzDipole:

    def __init__(self, r0, frequency, power):
        .
        .
        .

    def E_field(self, x, y, z, p, t):
        .
        .
        .
        return E
```

Zeitfenster angeben

```
n_t = 50
t_max = oscillating_dipoles[0].T
t = np.linspace(0, t_max, n_t)
```

Berechnung und Plotten des Feldes

```
phi = np.arctan2(yy, xx)
e_phi = np.array([-np.sin(phi), np.cos(phi)])
.
.
.
pos = 0
for dt in t:
    od = oscillating_dipoles[0]
    p = od.p_z * np.exp(-1j * od.omega * dt) * e_z
    Hx, Hy, Hz = util.total_field(xx, yy, zz,
                                  oscillating_dipoles,
                                  f='H_field',
                                  dynamic=True, p=p, dt=dt)
    .
    .
    .
    util.trim(Hx, Hy, cap=0.01)
    plt.quiver(xx, yy, Hx, Hy, np.array([Hx, Hy]) * e_phi, cmap='cool')
    anim.window(labels=[r'$x/$m', r'$y/$m'])
    anim.render_frame(t=t, loc='D_H', pos=pos)
    .
    .
    .
    pos += 1
```

6 Speichern als .gif-Datei

```
anim.render_anim(t=t, loc='D_H')
```

7 Link

https://github.com/jtormoehlen/EBFieldSolver