Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

SciPy und Pandas

Schriftliche Ausarbeitung eines Vortrags des Proseminars
"Python im Hochleistungsrechnen"
vom 20. Mai 2019

Von: Felix Maurer

Betreut durch: Jannek Squar

Inhaltsverzeichnis

1 Einführung **1**
1.1 Motivation 1
1.2 Vorgehensweise 1

2 SciPy **2**
2.1 Übersicht 2
2.2 Wichtige Submodule von SciPy 2
2.2.1 scipy.integrate 2
2.2.2 scipy.linalg 4
2.2.3 scipy.interpolate 5

3 Pandas **7**
3.1 Übersicht 7
3.2 Wichtige Funktionen von Pandas 7
3.2.1 pandas.Series 7
3.2.2 pandas.DataFrame 8
3.2.3 Pandas: I/O 9
3.3 Vergleich 9

4 Anwendungsbeispiel **11**

5 Literaturverzeichnis **III**

6 Anhang **1**

1 Einführung

1.1 Motivation

Nahezu alle Problemstellungen der (angewandten) Informatik im Allgemeinen und des High-Performance-Computings im Speziellen lassen sich in ihren Lösungen auf das Interpretieren, Verwalten und Verarbeiten von Daten abstrahieren.
Dabei variieren die Anforderungen an die Repräsentation dieser Daten zwischen unterschiedlichen Problemräumen immens. Während ein wenige Zeilen langes Skript beispielsweise mit einer kleinen Menge von Ganzzahlwerten oder Zeichenketten arbeitet, gibt es im Hochleistungsrechnen (aber auch im alltäglichen Gebrauch) Anwendungen, welche sehr viel größere Datenmengen handhaben und verwalten müssen.
Sofort ist klar, dass für letzteres ein effizientes und standardisiertes Verfahren zur Speicherung und Verarbeitung dieser Datenmengen vonnöten ist.
Für die Programmiersprache Python hat sich hier besonders eine Implementation durchgesetzt: Der `SciPy-Stack`. Dieser Umfasst zusätzlich zur `SciPy`-Pythonbibliothek selbst einige weitere Softwarepakete für Python, welche die oben beschriebenen Anforderungen an den Umgang mit Daten realisieren.

1.2 Vorgehensweise

Nebst der bereits Erwähnten `SciPy`-Pythonbibliothek werde ich mich dabei in dieser Ausarbeitung auf `Pandas`, einer weiteren Pythonbibliothek aus dem oben genannten `SciPy`-Stack, beschränken. Das bedeutet, dass ich zunächst den Zweck und weitere grundlegende Informationen zur `SciPy`-Bibliothek darlegen werde, um anschließend einige ihrer am häufigsten verwendeten Funktionen zu erläutern.
Ähnlich werde ich auch bei der Vorstellung von `Pandas` vorgehen, hierbei folgt jedoch abschließend ein anekdotischer Vergleich, um mögliche Vor- oder Nachteile dieses Softwarepaketes hinsichtlich Implementationskomplexität und Performanz aufzuzeigen.
Zu guter letzt werde ich ein Anwendungsbeispiel anführen, in welchem Funktionen von `SciPy` mit Funktionen von `Pandas` gekoppelt verwendet werden.

2 SciPy

2.1 Übersicht

`SciPy` ist ein Python-Modul, welches die grundlegenden numerischen Operationen Pythons um diverse weitere ergänzt. Auch wenn es sich hier um eine große Anzahl implementierter Funktionen handelt, impliziert der Name `SciPy` bereits, mit welchem Hintergedanken selektiert wurde: Der Anspruch des `SciPy`-Paketes ist es, insbesondere (numerische) Funktionen, welche bei wissenschaftlichem Rechnen (und Datenvisualisierung) frequent genutzt werden, effizient zu implementieren.
Erwähnt sei zudem, dass das `SciPy`-Modul genau so wie der restliche `SciPy`-Stack open-source und kostenlos nutzbar ist.
Mithilfe eines Paketmanagers wie `pip` lässt sich das Paket installieren (`$ pip install scipy`).

2.2 Wichtige Submodule von SciPy

Das `Scipy-Modul` ist hierarchisch aufgebaut und lässt sich daher in Submodule teilen. Insgesamt umfasst `SciPy` zwölf Submodule, ich werde mich jedoch auf folgende drei beschränken::

1. `scipy.integrate`
2. `scipy.linalg`
3. `scipy.interpolate`

Diese drei Submodule sollten helfen, die grundlegende Arbeitsweise und den Anwendungskontext von `SciPy` zu verstehen.

2.2.1 scipy.integrate

Insbesondere in der Naturwissenschaft, aber auch in allen anderen empirischen Wissenschaften ist die Integration eine häufig verwendete mathematische Funktion. Klar ist daher, dass die Zielgruppe von `SciPy` (d.h. wissenschaftliche Python-Anwender) diese standardmäßig nicht in Python implementierte Funktion nutzen möchten.
Die Schnittstelle von `scipy.integrate` werde ich daher anhand zweier grundlegender

Integrationsfunktionen erläutern: Dem einfachen und dem doppelten Integral.

Das einfache Integral: **integrate.quad**

Die `SciPy`-Funktion `quad` benutzt zur Berechnung des Integrals der Funktion `func` mit Untergrenze `a` und Obergrenze `b` die Syntax:

```
scipy.integrate.quad(func, a, b).
```

Beispiel: $\int_0^5 x^2 dx$

```
In [15]: import scipy.integrate

In [13]: scipy.integrate.quad(lambda x: x**2,0,5)
Out[13]: (41.66666666666666, 4.625929269271485e-13)
```

Abbildung 1: Anwendung von `scipy.integrate.quad`

Das doppelte Integral: **integrate.dblquad**

Analog zum einfachen Integral berechnet `dblquad` ein Doppelintegral. Hierbei sei erneut `func` die zu integrierende Funktion und `a` und `b` die äußere Unter- und Obergrenze, jedoch seien zudem `gfunc` und `hfunc` zwei Funktionen, die die Unter- und Obergrenze des inneren Integrals beschreiben. Dann bedeutet dass für `dblquad`:

```
scipy.integrate.dblquad(func, a, b, gfun, hfun)
```

Beispiel: $\int_{y=0}^{1} \int_{x=0}^{2y} (x+y) dx dy$

```
In [15]: import scipy.integrate

In [17]: scipy.integrate.dblquad(lambda x,y: x+y,0,1, lambda x: 0, lambda x: 2*x)
Out[17]: (1.3333333333333335, 4.421626967161769e-14)
```

Abbildung 2: Anwendung von `scipy.integrate.dblquad`

2.2.2 scipy.linalg

Viele datenverarbeitenden Operationen lassen sich auf Operationen an Matrizen zurückführen. Daher ist eine effiziente Implemtation elementarer Matrizenoperationen mit einfacher Nutzerschnittstelle etwas, das `SciPy` ebenfalls bietet. Gesammelt sind diese Funktionen im Submodul `scipy.linalg`.
Im folgenden werde ich erläutern, wie man hiermit eine Matrix invertieren kann, wie man ihre Determinante bildet und wie man die Lösungsmenge eines linearen Gleichungssystems ermittelt.

Inverses und Determinante einer Matrix: **linalg.inv und linalg.det**
Beide Funktionen benötigen jeweils eine Matrix quadratische `A` als übergebenen Parameter. Der Funktionsaufruf sieht dann jeweils wie folgt aus:

```
scipy.linalg.inv(A)
scipy.linalg.det(A)
```

Beispiel:

Lineares Gleichungssystem lösen: **linalg.solve**

Die Funktion `solve` bestimmt den Vektor x eines Gleichungssystem der Form $A \times x = b$.

Syntax:

```
scipy.linalg.solve(A, b)
```

Beispiel:

```
In [50]: import scipy.linalg

In [58]: A= np.array([[4,2],[3,1]])
         b= np.array([[1],[2]])
         print(A)
         print('.....')
         print(b)

         [[4 2]
          [3 1]]
         .....
         [[1]
          [2]]

In [59]: scipy.linalg.solve(A,b)

Out[59]: array([[ 1.5],
                [-2.5]])
```

Abbildung 3: Anwendung von `scipy.linalt.solve`

2.2.3 scipy.interpolate

Interpolationsfunktionen beschäftigen sich damit, zu einer endlichen Menge von Zuordnungen eines Elements aus einer Definitionsmenge zu einem Element einer Abbildungsmenge durch Annäherungsverfahren eine stetige Funktion zu approximieren.

In `SciPy` befinden sich solche Interpolationsfunktionen im Submodul `scipy.interpolate`.

Ich beschränke mich hierbei auf die grundlegendste und populärste Funktion des Submoduls: `interp1d`.

Es seien x und y zwei Datenstrukturen mit gleicher Anzahl Elementen. Dann kann durch Aufruf von

```
scipy.interpolate.interp1d(x, y, kind)
```

eine stetige Funktion approximiert werden, wobei angenommen wird, dass das n-te Element aus x stets auf das n-te Element aus y abgebildet wird.
Der optionale Parameter *kind* beeinflusst dabei die Art der Approximation. Zur Auswahl stehen beispielsweise linear, quadratisch und kubisch.

Beispiel:

```
In [233]: import numpy as np
          import matplotlib.pyplot as plt
          import scipy.interpolate

In [239]: x = [0,1,2,3,4,5,6,7,8,9]
          y = [0.0, 0.841, 0.909, 0.141, -0.757,
               -0.959, -0.279, 0.657, 0.989, 0.412]

          f = interp1d(x, y, kind='linear')
          f2 = interp1d(x, y, kind='quadratic')

In [240]: xnew = np.linspace(0, 9, num=15)
          plt.plot(x, y, 'o', xnew, f(xnew), '--', xnew, f2(xnew), '-')
          plt.show()
```

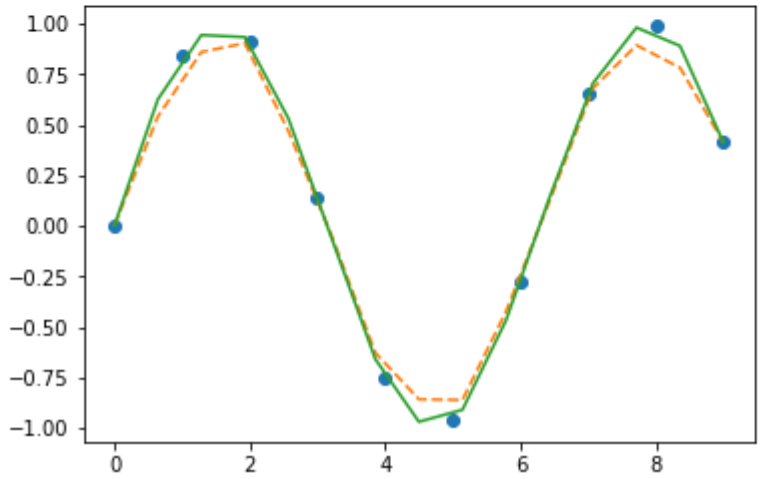

Abbildung 4: Anwendung von `scipy.interpolate.interp1d`

3 Pandas

3.1 Übersicht

Als zweiter Teil des Vortrags beschäftigte ich mit einem weiteren Python-Modul des `SciPy`-Stacks. Dieses ermöglicht effiziente, tabellarische Verwaltung von Daten in Python und nennt sich `Pandas`. Der Name leitet sich dabei aus *panel data*, der englischen Bezeichnung für diese Art der Datenverwaltung, ab.

`Pandas` bietet damit eine Alternative zum bisher genutzten und ebenfalls dem `SciPy`-Stack entstammenden `NumPy`.
Dabei bietet `Pandas` quasi eine Erweiterung von letzterem, da es Datenstrukturen aus `Numpy` problemlos versteht und somit kompatibel ist.

Ebenso wie der Rest des `SciPy`-Stacks ist `Pandas` open-source und kostenlos, analog zu `SciPy` lässt es sich beispielsweise mit `pip` (`$ pip install pandas`) herunterladen.

3.2 Wichtige Funktionen von Pandas

3.2.1 pandas.Series

Als erstes betrachten wir die `Series`. Diese funktioniert ähnlich einem `NumPy`-Array, kann jedoch um Bezeichner, welche anstelle von Indizes treten, erweitert werden, nach diesen filtern und diese auch nachträglich bearbeiten.
Wie dies konkret aussieht, wird im nächsten Unterpunkt klar, da dort die für `DataFrames` identische Vorgehensweise erläutert wird.
Erstellen lassen sich `Series`-Elemente durch

```
pandas.Series(data, index),
```

wobei *data* und *index* zwei identisch große Datenstrukturen sein müssen. In *index* müssen sich Zeichenketten oder primitive Datentypen befinden.

3.2.2 pandas.DataFrame

Ein `DataFrame` speichert Daten tabellarisch. Das bedeutet, dass man einen zweidimensionalen Datensatz hat, welcher durch Zeilen- und Spaltenbezeichner strukturiert wird. Sei beispielsweise ein $n \times m$-Datensatz gegeben, dann muss die zeilenbezeichnende Datenstruktur (z.B eine Liste von Zeichenketten) genau m und die spaltenbezeichnende Datenstruktur genau n Elemente behinhalten.
Initialisiert wird ein neuer *DataFrame* mit Datensatz *data*, Zeilenbezeichner *index* und Spaltenbezeichner *columns* durch

`pandas.DataFrame(data, index, columns)`.

Grundlegende Funktionen von DataFrames
Sei `df` ein DataFrame. Dann sind Grundlegende Operationen auf *df*:

- `df.to_numpy()`: Konvertierung von DataFrame zu `NumPy`-Array
- `df[a], df.a`: Spalte mit Titel 'a' als Series
- `df[a:b]`: Zeilen zwischen a (inklusiv) und b (exklusiv)
- `df.head(a)`: Alle Zeilen bis Index a (exklusiv); standardmäßig 5
- `df.tail(a)`: Die letzten a Zeilen von *df* als DataFrame; standardmäßig 5
- `df.sort_values(a)`: Zeilen von *df* werden aufsteigend nach Spalte a sortiert.
- `df.columns`: Gibt Spaltenbezeichner aus.
- `df.index`: Gibt Zeilenbezeichner aus.
- `df.loc[a,b]`: In Zeile(n) 'a' und Spalte(n) 'b'
- `df.iloc[a,b]`: In Zeile(n) a und Spalte(n) b
- `df.ix[a,b]`: Hybrid aus `loc` und `iloc`

- `df.at[a,b] = c`: Element(e) bei [a,b] wird/werden durch c ersetzt.
- `df.apply(func)`: Für jedes Element a ∈ df wird ein func(a) gegeben.
- `df.dropna()`: Entfernt eine Zeile, wenn in ihr Einträge NaN sind.
- `df.fillna(a)`: Ersetzt jedes NaN in df durch a.
- `df.append(a)`: Fügt alle Zeilen aus a (DataFrame) df hinzu.
- `pandas.concat([a,b,c])`: Erstellt aus a, b und c einen DataFrame.

3.2.3 Pandas: I/O

`Pandas` kann diverse Dateiformate lesen und zur Laufzeit gespeicherte Daten in ebendiese Dateiformate konvertieren. Mögliche Dateiformate sind dabei unter Anderem `.scv`, `.html` und `json`.
Auch mit bei Software wie Microsoft Excel oder SQL verwendeten Datentypen ist `Pandas` kompatibel.

3.3 Vergleich

Wie bereits angekündigt wird nun eine `Pandas`-Implementation mit einer Vanilla-Python-Implementation (d.h ein Pythonskript, in welchem keine externen Module geladen werden) verglichen.
Anekdotisch wird die Performanzdifferenz und Unterschied im Implementationsaufwand durch Verwendung von `pandas.DataFrame.iloc` gezeigt.
Beide Implementationen befinden sich im Anhang.
Direkt zeigt sich, dass das zweidimensionale Selektieren aus einem verschachtelten Datensatz in Python nur über unschöne Umwege gelöst wurde. Wird beispielsweise eine weitere Farbe nachträglich hinzugefügt, ist dies insofern problematisch, als dass man den Quelltext der `funcPython`-Funktion abwandeln müsste. Die Meinung, dass die Impementation mit `Pandas` daher schöner ist, sollte daher nachvollziehbar sein.
Wenn man nun mit *%timeit* einen Performance-Vergleich durchführt, kommt man auf folgendes Ergebnis:

```
In [19]: %timeit funcPandas(0,1)
         7.42 µs ± 37.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [20]: %timeit funcPython(0,1)
         108 ns ± 0.14 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Abbildung 5: Anwendung von *%timeit*.

Es zeigt sich also, dass es sich (bezogen auf die Performanz) bei einer so kleinen Datenmenge nicht lohnt, mit `Pandas` zu arbeiten.
Wie sich schon beim Implementationsvergleich zeigte, skalieren `Pandas`-Lösungen jedoch sehr viel besser mit größer werdenden Datenmengen. Dies trifft ebenfalls im Hinblick auf Performance zu.

4 Anwendungsbeispiel

Da mein Vortrag zum Thema mit einem Anwendungsbeispiel abschloss, werde ich dieses nun schrittweise erläutern.
Die Implementationen und bei Ausführung entstandenen Ausgaben sind im Anhang zu finden.
Das Anwendungsbeispiel beginnt mit mehreren `import`-Befehlen. Konkret werden `NumPy`, `Pandas`, `scipy.interpolate` und Teile von `matplotlib` importiert.
Anschließend wird ein `pandas.DataFrame` erzeugt, indem mit `pandas.read_csv` ein Datensatz eingelesen wird. Es handelt sich bei dem Datensatz um Koordinaten der Erde im Verlauf von 400 Tagen, jedoch fehlen einige Daten.
Im soeben erzeugten `DataFrame` werden anschließend die Spalten umbenannt.

Nun werden mithilfe von `matplotlib` parallel alle drei Koordinatenwerte (X, Y und Z) in ihrer zeitlichen Entwicklung geplottet. Zudem wird durch `scipy.interpolate.interp1d` versucht durch Interpolation die fehlenden Werte zu schätzen.

Standardmäßig interpoliert `interp1d` linear. Da dies aber für die Annäherung der fehlenden X- und Y-Werte nicht sinnvoll ist, wird anschließend durch hinzufügen des optionalen Parameters *quadratic* quadratisch approximiert, was zu realistischeren Ergebnissen führt.

Abschließend wird nun angenommen, die fehlenden Datensätze seien nun doch aufgetaucht. Es lässt sich durch `pandas.DataFrame.append` nun mühelos alles Fehlende hinzufügen.

5 Literaturverzeichnis

Alle Websites wurden zuletzt am 19.05.2019 aufgerufen.

- "The SciPy Stack Specification", https://www.scipy.org/stackspec.html
- "SciPy", https://scipy.github.io/devdocs
- Eric van Rees: "pandas and NumPy arrays explained", https://medium.com/@ericvanrees/pandas-series-objects-and-numpy-arrays-15dfe05919d7
- "pandas: powerful Python data analysis toolkit", https://pandas.pydata.org/pandas-docs/stable/
- Shane Lynn: Üsing iloc, loc, ix to select rows and columns in Pandas DataFrames", https://www.shanelynn.ie/select-pandas-dataframe-rows-and-columns-using-iloc-loc-and-ix/
- Wikipedia: "NumPy", https://de.wikipedia.org/wiki/NumPy
- Wikipedia: "SciPy", https://de.wikipedia.org/wiki/SciPy
- Wikipedia: "Pandas (Software)", https://de.wikipedia.org/wiki/Pandas_(Software)
- Dharmindar Devsidas: "Python Scientific Programming - SciPy Basic Interpolation", https://www.youtube.com/watch?v=b0jo2g3iKJM
- John Foster: "SciPy - Scientific Toolkit", https://www.youtube.com/watch?v=MtdLd2lrvag
- Harrison Kinsley: "Data Analysis with Python and Pandas Tutorial Introduction", https://www.youtube.com/watch?v=Iqjy9UqKKuo
- "Profiling and Timing Code", https://jakevdp.github.io/PythonDataScienceHandbook/01.07-timing-and-profiling.html

6 Anhang

In [12]:
```
import pandas as pd
import numpy as np
s = pd.DataFrame([[1,11],[2,22],[3,33]],index=['rot', 'gelb', 'gruen'])

def funcPandas(x,y):
    return s.iloc[x,y]

funcPandas(0,1) #example
```

Out[12]: 11

In [13]:
```
dict = {
  "rot": [1,11],
  "gelb": [2,22],
  "gruen": [3,33],
}

def funcPython(x,y):
    if x==0:
        relevant=dict["rot"]
    elif x==1:
        relevant=dict["gelb"]
    elif x==2:
        relevant=dict["gruen"]
    return relevant[y]

funcPython(0,1) #example
```

Out[13]: 11

Abbildung 6: Implementation mit und ohne `Pandas`

```
In [82]: import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
         import scipy.interpolate as ip

In [94]: data = pd.read_csv('/home/felix/Documents/astrodata', sep=" ")
         data.columns = ['Day', 'X', 'Y', 'Z']
```

Abbildung 7: Anwendungsbeispiel (1)

```
In [89]: fx = ip.interp1d(data.Day,data.X)
         fy = ip.interp1d(data.Day,data.Y)
         fz = ip.interp1d(data.Day,data.Z)

         days = np.arange(0, 400)

         plt.plot(data.Day,data.X,'o',
                  data.Day,data.Y,'o',
                  data.Day,data.Z,'o',
                  days,fx(x), days,fy(x), days,fz(x))
         plt.show()
```

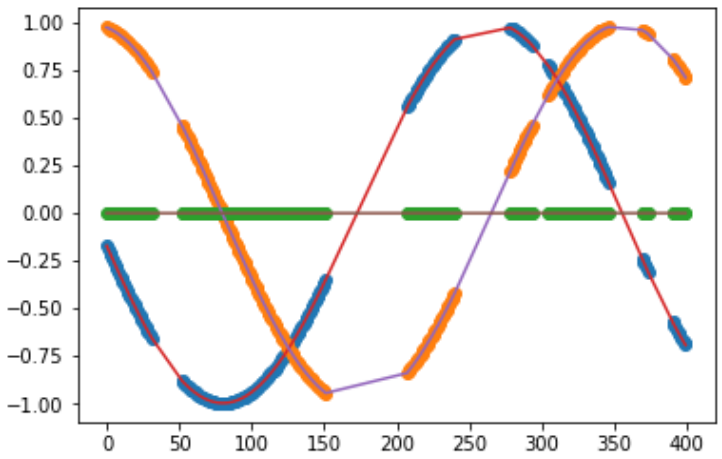

Abbildung 8: Anwendungsbeispiel (2)

```
In [90]: fx = ip.interp1d(data.Day,data.X, kind='quadratic')
         fy = ip.interp1d(data.Day,data.Y, kind='quadratic')
         fz = ip.interp1d(data.Day,data.Z)

         days = np.arange(0, 400)

         plt.plot(data.Day,data.X,'o',
                  data.Day,data.Y,'o',
                  data.Day,data.Z,'o',
                  days,fx(x), days,fy(x), days,fz(x))
         plt.show()
```

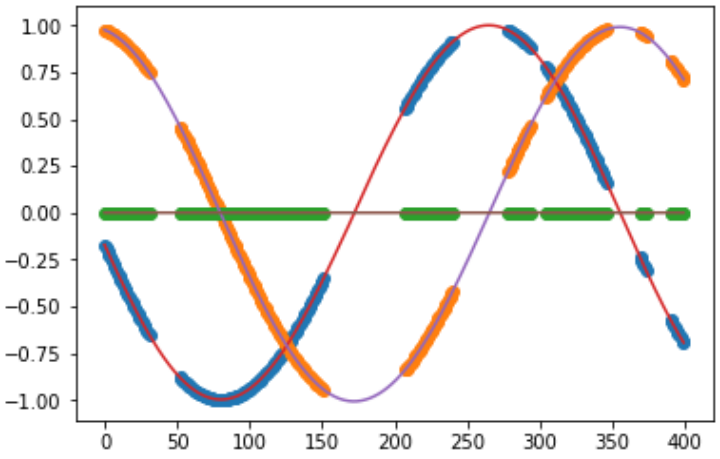

Abbildung 9: Anwendungsbeispiel (3)

```
In [91]: data = data.append(pd.read_csv('/home/felix/Downloads/ss19_2.txt', sep=" "))
         plt.plot(data.Day,data.X,'o', data.Day,data.Y,'o', data.Day,data.Z,'o')
         plt.show()
```

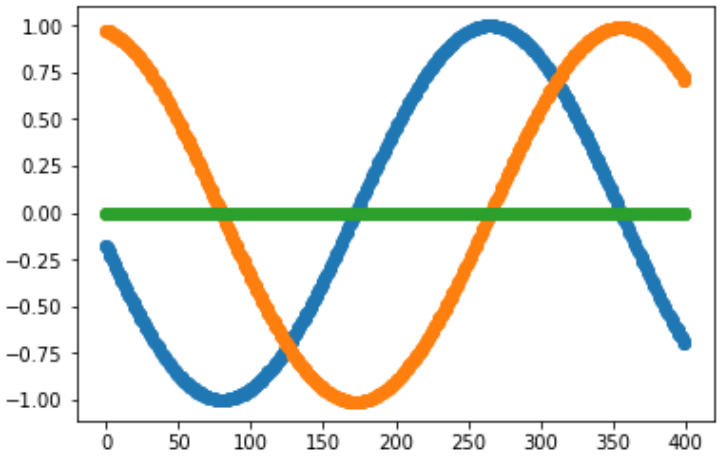

Abbildung 10: Anwendungsbeispiel (4)