SciPy_Tuebix_2015

June 18, 2015

1 Einführung in SciPy und SymPy

Peter Hrenka, TÜBIX 2015

Repository auf github

1.1 1. IPython Grundlagen

IPython ist eine interaktive Shell für Python. Startet man IPython mit

> `ipython notebook`

wird ein lokaler Webserver gestartet, so daß man mit einem Browser neue Notebooks erstellen und editieren kann.

Es wird das asynchrone Python-Webserver tornado verwendet, der eine Websocket-Verbindung mit dem Webbrowser herstellt.

Für die Darstellung mathematischer Formeln wird MathJax verwendet, welches einen nützlichen Teil der LaTeX-Syntax versteht.

Das Notebook wird über ipython zu einem `Kernel` verbunden, der in einem separaten laufenden Prozess läuft und die Python-Befehle ausführt. Es sind auch Kernel für Julia, R und Clang und viele andere vorhanden.

Ein Notebook besteht aus mehreren "Zellen" (Cells) die Python-Code oder Markdown-formatierten Text enthalten können.

```
In [4]: import numpy as np
        # from numpy import *
        print(np.__version__)
        np
```

```
1.9.2
```

```
Out[4]: <module 'numpy' from '/home/speter/anaconda/lib/python2.7/site-packages/numpy/__init__.pyc'>
```

Code-Zellen enthalten eine auch mehrzeilige Eingabe.

Ausgeführt wird der Code in der Zelle mit `CTRL + RETURN`.

Eine neue Zelle erhält man mit `ALT + RETURN`.

Eventuelle Ausgaben auf `sys.stdout` oder `sys.stderr` erscheinen direkt nach der Eingabebereich, und der letzte Ausdruck wird im gesonderten Ausgabebereich angezeigt.

Wie `vi` unterscheidet `ipython` einen Kommando- und Edit-Modus: * Im Kommando-Modus kann man mit den Pfeiltasten die Zellen wechseln, und etwa mit `dd` die aktelle Zelle löschen * Mit `RETURN` oder Maus-Doppelklick kommt man in den Edit-Modus, den man mit `ESC` oder `CTRL + RETURN` verlassen kann

```
In [5]: # Interaktive Hilfe
        np?
```

1.2 2. numpy

`numpy` ist eine Python-Bibliothek die effiziente Arrays und darauf operierende Funktionen, die in C, C++ oder FORTRAN implementiert sind, in Python verfügbar macht. Sie steht unter der BSD-Lizenz. Mittlerweile ist sie (konzeptionell) ein Bestandteil von SciPy, kann aber meist separat installiert werden.

`numpy` ist recht umfangreich und leider nicht sehr gut modularisiert, so dass man sich beim ersten Start nicht über einige Gedenksekunden wundern sollte.

1.2.1 Das Klasse `array`

Das zentrale Objekt in `numpy` ist das `array` (in der Dokumentation auch oft noch `ndarray` genannt). Es ist ein mehrdimensionaler Container für Elemente des gleichen Datentyps.

```
In [6]: np.array?
```

```
In [7]: a = np.array([1,2,3,4], dtype=np.float32)
        a
```

```
Out[7]: array([ 1.,  2.,  3.,  4.], dtype=float32)
```

```
In [8]: a.shape, a.dtype
```

```
Out[8]: ((4,), dtype('float32'))
```

```
In [9]: a.flags
```

```
Out[9]:   C_CONTIGUOUS : True
          F_CONTIGUOUS : True
          OWNDATA : True
          WRITEABLE : True
          ALIGNED : True
          UPDATEIFCOPY : False
```

```
In [10]: b = a.reshape( (2,2) )
         b
```

```
Out[10]: array([[ 1.,  2.],
                [ 3.,  4.]], dtype=float32)
```

```
In [11]: b.flags
```

```
Out[11]:   C_CONTIGUOUS : True
           F_CONTIGUOUS : False
           OWNDATA : False
           WRITEABLE : True
           ALIGNED : True
           UPDATEIFCOPY : False
```

Man hieran erkennen, dass `a` eigene Daten besitzt (`OWNDATA: True`) und `b` nicht. Das spart natürlich Speicher, kann aber auch leicht zu schwer lokalisierbaren Fehlern führen, denn bei Änderungen an einem Objekt ändert sich auch das andere.

```
In [12]: c = np.array( np.reshape(a, (2,2)) )
         c.flags.owndata
```

```
Out[12]: True
```

Lineare Bereiche kann man mit `linspace(start, stop, num)` erstellen. Man beachte, dass der Parameter `num` einen etwas willkürlichen Standardwert von 50 hat, und man daher besser immer die gewünsche Anzahl eingeben sollte.

```
In [13]: lin = np.linspace(1.0, 42.0, 42)
         lin
```

```
Out[13]: array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.,
                12., 13., 14., 15., 16., 17., 18., 19., 20., 21., 22.,
                23., 24., 25., 26., 27., 28., 29., 30., 31., 32., 33.,
                34., 35., 36., 37., 38., 39., 40., 41., 42.])
```

Arrays können über eckige Klammern indiziert werden. Pro Dimension kann die Slice-Syntax wie bei Python-Listen verwendet werden, `a[start:stop]` oder `a[start:stop:step]`. Der Wert dieses Ausdrucks ist wieder ein Array der entsprechenden Dimension. Solche Ausdrücke können auch das Ziel einer Zuweisung sein.

```
In [14]: rect = np.reshape(lin, (6,7))
         rect
```

```
Out[14]: array([[ 1.,  2.,  3.,  4.,  5.,  6.,  7.],
                [ 8.,  9., 10., 11., 12., 13., 14.],
                [15., 16., 17., 18., 19., 20., 21.],
                [22., 23., 24., 25., 26., 27., 28.],
                [29., 30., 31., 32., 33., 34., 35.],
                [36., 37., 38., 39., 40., 41., 42.]])
```

```
In [15]: # Alle Einträge von Zeile 1
         rect[1, :]
```

```
Out[15]: array([ 8.,  9., 10., 11., 12., 13., 14.])
```

```
In [16]: # jeder zweite Eintrag von Spalte 3
         rect[::2, 2]
```

```
Out[16]: array([ 3., 17., 31.])
```

```
In [17]: cuboid = np.reshape(lin, (2,3,7))
         cuboid
```

```
Out[17]: array([[[ 1.,  2.,  3.,  4.,  5.,  6.,  7.],
                 [ 8.,  9., 10., 11., 12., 13., 14.],
                 [15., 16., 17., 18., 19., 20., 21.]],

                [[22., 23., 24., 25., 26., 27., 28.],
                 [29., 30., 31., 32., 33., 34., 35.],
                 [36., 37., 38., 39., 40., 41., 42.]]])
```

Zusätzlich zu konkreten Zahlen und Slices kann bei `arrays` auch die `Ellipsis` ... verwendet werden (die Offenbar nur für `numpy` in die Python-Grammatik eingebaut wurde). Diese entspricht der maximalen möglichen Anzahl an `:`-Slices und darf auch nur einmal vorkommen (analog zu `::` in IPv6).

```
In [18]: cuboid[1,...]
```

```
Out[18]: array([[22., 23., 24., 25., 26., 27., 28.],
                [29., 30., 31., 32., 33., 34., 35.],
                [36., 37., 38., 39., 40., 41., 42.]])
```

```
In [19]: cuboid[...,0]
```

```
Out[19]: array([[ 1.,  8., 15.],
                [22., 29., 36.]])
```

1.2.2 Arithmetik mit `arrays`

```
In [20]: lin + 1
```

```
Out[20]: array([ 2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.,
                13., 14., 15., 16., 17., 18., 19., 20., 21., 22., 23.,
                24., 25., 26., 27., 28., 29., 30., 31., 32., 33., 34.,
                35., 36., 37., 38., 39., 40., 41., 42., 43.])
```

Was ist da passiert? Die `1` wurde zu jedem Array-Element hinzuaddiert!

Das Verhalten nennt sich `broadcasting`: Numpy versucht, die Dimension(en) der Operanden aneinander anzugleichen.

In diesem Fall wird die `1` als array aufgefasst und die Dimension auf `42` erweitert.

```
In [21]: np.broadcast(lin, 1).shape
```

```
Out[21]: (42,)
```

```
In [22]: lin*lin
```

```
Out[22]: array([  1.00000000e+00,   4.00000000e+00,   9.00000000e+00,
                  1.60000000e+01,   2.50000000e+01,   3.60000000e+01,
                  4.90000000e+01,   6.40000000e+01,   8.10000000e+01,
                  1.00000000e+02,   1.21000000e+02,   1.44000000e+02,
                  1.69000000e+02,   1.96000000e+02,   2.25000000e+02,
                  2.56000000e+02,   2.89000000e+02,   3.24000000e+02,
                  3.61000000e+02,   4.00000000e+02,   4.41000000e+02,
                  4.84000000e+02,   5.29000000e+02,   5.76000000e+02,
                  6.25000000e+02,   6.76000000e+02,   7.29000000e+02,
                  7.84000000e+02,   8.41000000e+02,   9.00000000e+02,
                  9.61000000e+02,   1.02400000e+03,   1.08900000e+03,
                  1.15600000e+03,   1.22500000e+03,   1.29600000e+03,
                  1.36900000e+03,   1.44400000e+03,   1.52100000e+03,
                  1.60000000e+03,   1.68100000e+03,   1.76400000e+03])
```

```
In [23]: np.sin(lin)
```

```
Out[23]: array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 , -0.95892427,
                -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849, -0.54402111,
                -0.99999021, -0.53657292,  0.42016704,  0.99060736,  0.65028784,
                -0.28790332, -0.96139749, -0.75098725,  0.14987721,  0.91294525,
                 0.83665564, -0.00885131, -0.8462204 , -0.90557836, -0.13235175,
                 0.76255845,  0.95637593,  0.27090579, -0.66363388, -0.98803162,
                -0.40403765,  0.55142668,  0.99991186,  0.52908269, -0.42818267,
                -0.99177885, -0.64353813,  0.29636858,  0.96379539,  0.74511316,
                -0.15862267, -0.91652155])
```

Im Regelfall sind alle Operationen in `arrays` komponentenweise.

1.3 Übung (Arrays)

a) Fülle das 2d-array mit einem Schachbrettmuster, wobei Schwarz=0, Weiß=1
b) Schreibe 2 auf die Positionen der Diagonale

```
In [24]: # Schachbrett
         chess = np.zeros((10,10))
         chess
```

```
Out[24]: array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

1.4 Beispiel: Die Mandelbrot-Menge

Betrachte die Folgen $z_{n+1} = z_n^2 + c$ für alle $c \in \mathbb{C}$, $-2 < \mathrm{Re}\, c < 1$, $-1 < \mathrm{Im}\, c < 1$

Die 'Mandelbrot-Menge' ist die Menge aller c, für die z_n beschränkt bleibt.

Eine Python-Funktion, welche die Folge berechnet, kann man so schreiben:

```
In [25]: def mandel_series(c, numIter):
             series = np.zeros(shape=(numIter,), dtype=np.complex128)
             z = 0+0j
             for i in range(numIter):
                 z = z**2 + c
                 series[i] = z
             return series

         mandel_series(c=0.5+0.4j, numIter=10) # numIter=20
```

```
Out[25]: array([  5.00000000e-01 +4.00000000e-01j,
                  5.90000000e-01 +8.00000000e-01j,
                  2.08100000e-01 +1.34400000e+00j,
                 -1.26303039e+00 +9.59372800e-01j,
                  1.17484960e+00 -2.02343400e+00j,
                 -2.21401359e+00 -4.35446125e+00j,
                 -1.35594766e+01 +1.96816728e+01j,
                 -2.03008838e+02 -5.33346361e+02j,
                 -2.43245252e+05 +2.16548450e+05j,   1.22750214e+10 -1.05348765e+11j])
```

Das funktioniert, ist aber wenig anschaulich.

1.5 3. matplotlib

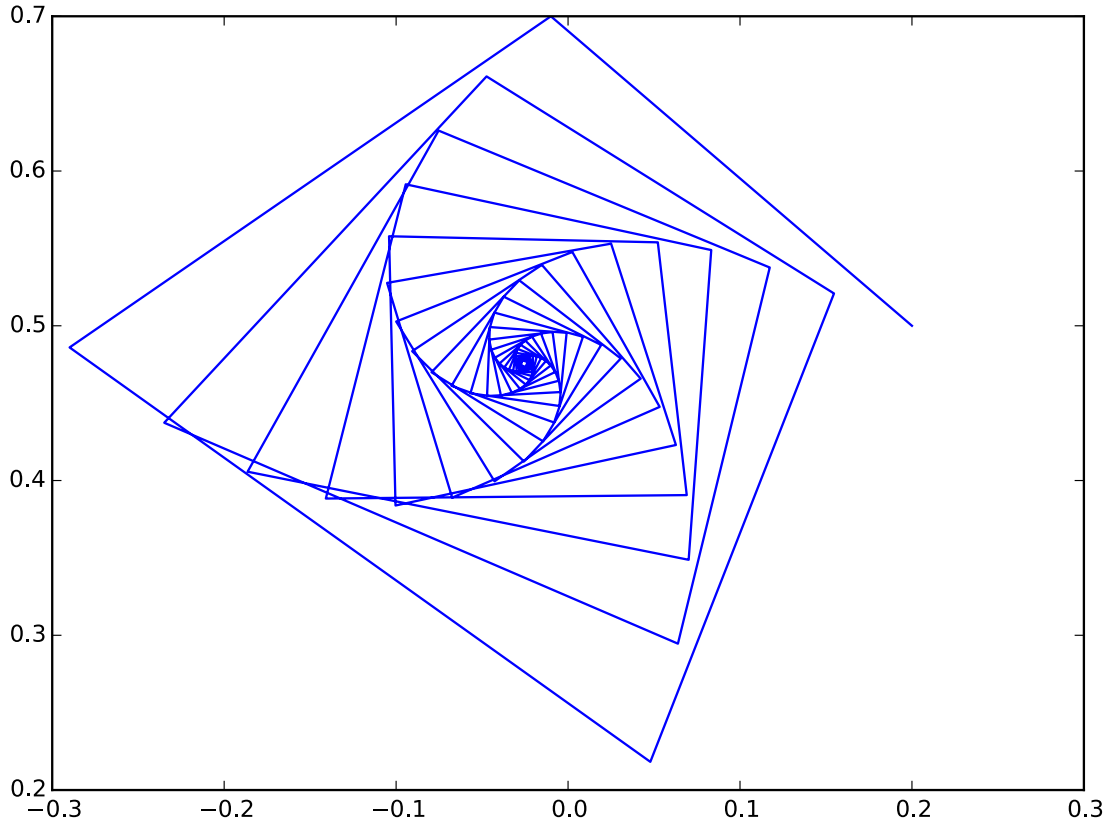
Im `ipython`-Umfeld ist es sinnvoll, `matplotlib` über die zugehörige Direktive einzubinden und einige Abkürzungssymbole zu definieren.

Matplotlib ist eine Python-Bibliothek, die unter einer BSD-artigen Lizenz steht und eine Reihe von Backends (wxPython, GTK+, Qt, ...) unterstützt. Sie bietet eine ähnliche Funktionalität wie Gnuplot an, verwendet aber Python-Syntax und hat eine Reihe von Abhängigkeiten, zuvorderst `numpy`.

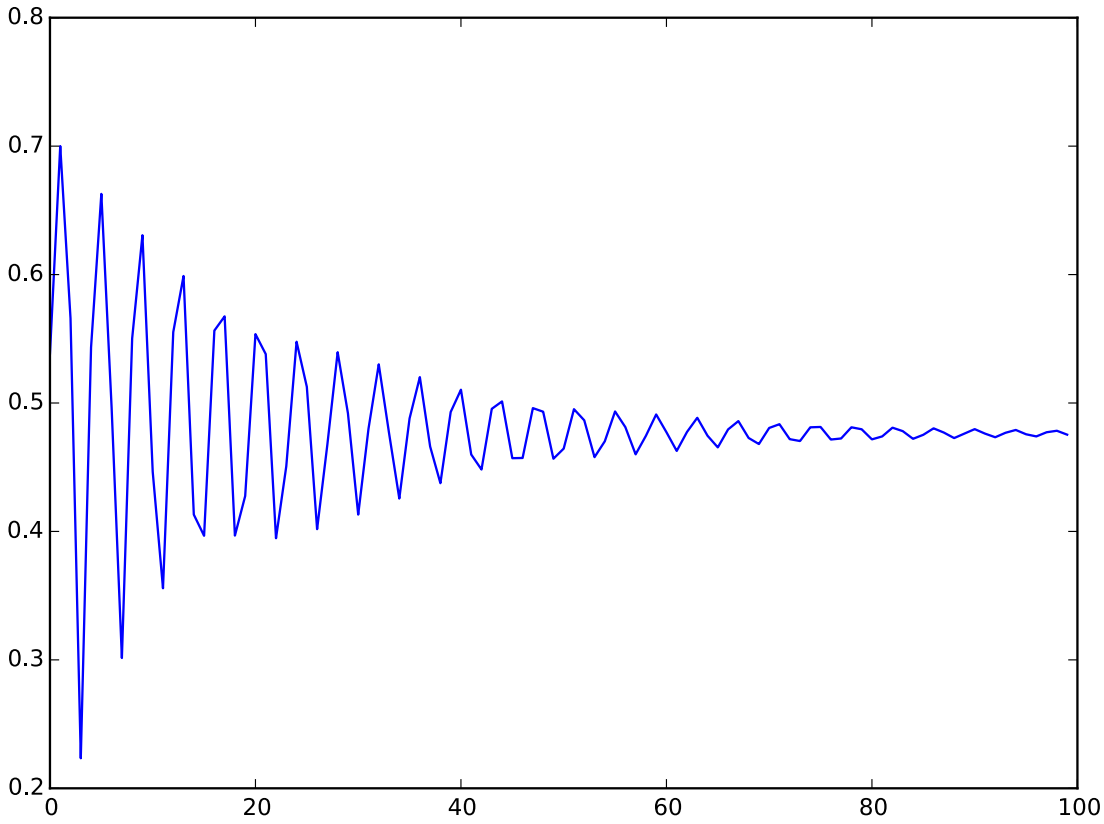
```
In [26]: %matplotlib inline
         %config InlineBackend.figure_format = 'svg'

         import matplotlib as mp
         import matplotlib.pyplot as plt

         plt.rcParams['figure.figsize'] = 8,6 # Größe in Inches (!!!)
```

Die Funktion `plt.plot` generiert einen Plot, der direkt im Notebook angezeigt wird.

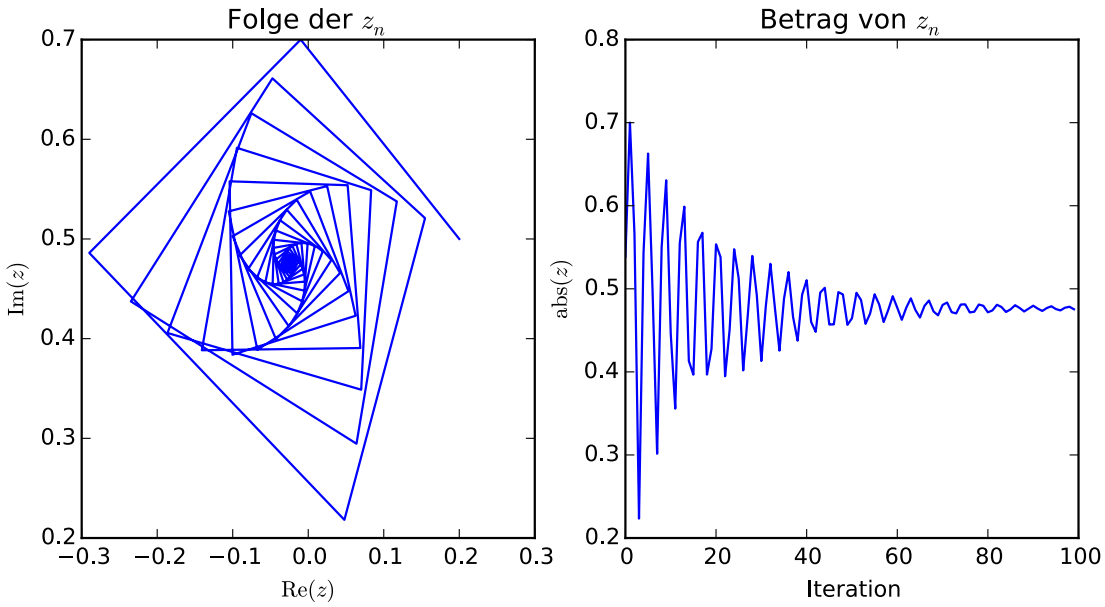
In [27]:
```
s1 = mandel_series(0.2+0.5j, 100)
plt.plot(s1.real, s1.imag)
plt.show()
```



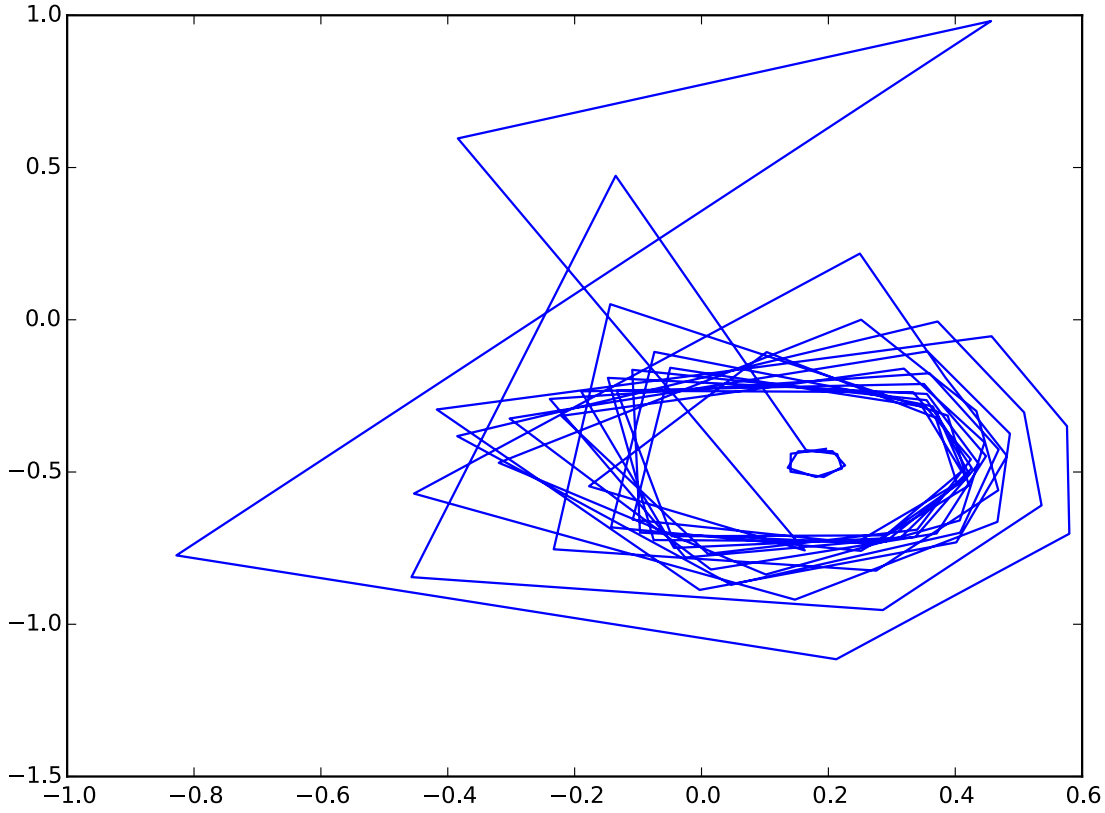
In [28]:
```
plt.plot(np.abs(s1));
```



In [29]:
```
s1 = mandel_series(0.2+0.5j, 100)

    plt.figure(figsize=(8,4))
    plt.subplot(1,2,1) # nrows, ncols, plot_number
    plt.title("Folge der $z_n$")
    plt.plot(s1.real, s1.imag)
    plt.xlabel("$\mathrm{Re} (z)$")
    plt.ylabel("$\mathrm{Im} (z)$")

    plt.subplot(1,2,2)
    plt.title("Betrag von $z_n$")
    plt.plot(np.abs(s1))
    plt.xlabel("Iteration")
    plt.ylabel("$\mathrm{abs} (z)$")

    plt.show()
```




```
In [30]: s1 = mandel_series(0.37-0.3j, 100) # oder vielleicht 0.39-0.3j ?
         plt.plot(s1.real, s1.imag)
         plt.show()
```



Damit können wir individuell feststellen, welche cs zur Mandelbrot-Menge gehören.
Aber vielleicht erkennt man ja ein Muster, wenn man mehrere c gleichzeitig berechnet...

```
In [31]: c = np.linspace(-2, 1, 800)+np.linspace(-1, 1, 600).reshape(600,1)*(0+1j)
         c.shape, c[0,0], c[599,799]
```

```
Out[31]: ((600, 800), (-2-1j), (1+1j))
```

```
In [32]: def mandel(c):
             z = np.zeros_like(c)
             for i in range(10):
                 z = z**2 + c
             return z
         z = mandel(c)
         z[0,0]
```

```
Out[32]: (-9.2568536143635449e+133+9.3386771787027272e+133j)
```

1.5.1 Performancemessung in `ipython`

Mit der `%timeit` Direktive kann man in IPython einfach die Geschwindigkeit einer Funktion messen:

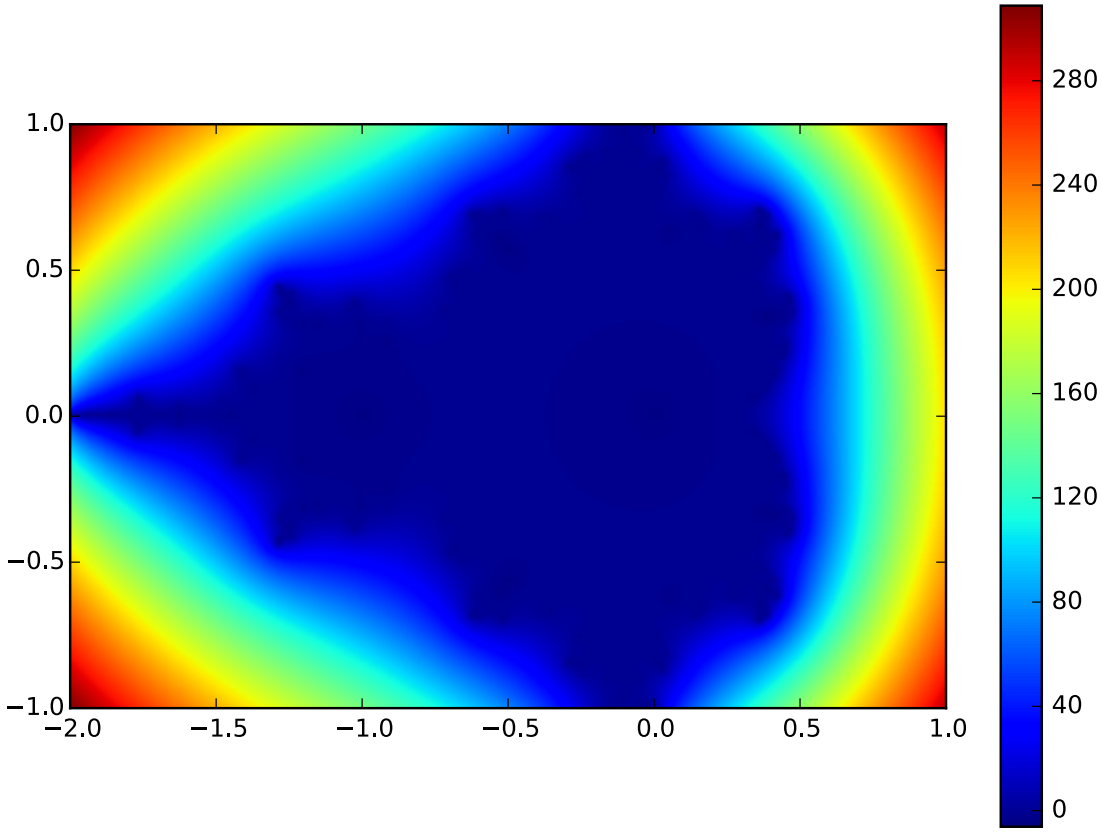
```
In [33]: %timeit mandel(c)
```

```
10 loops, best of 3: 165 ms per loop
```

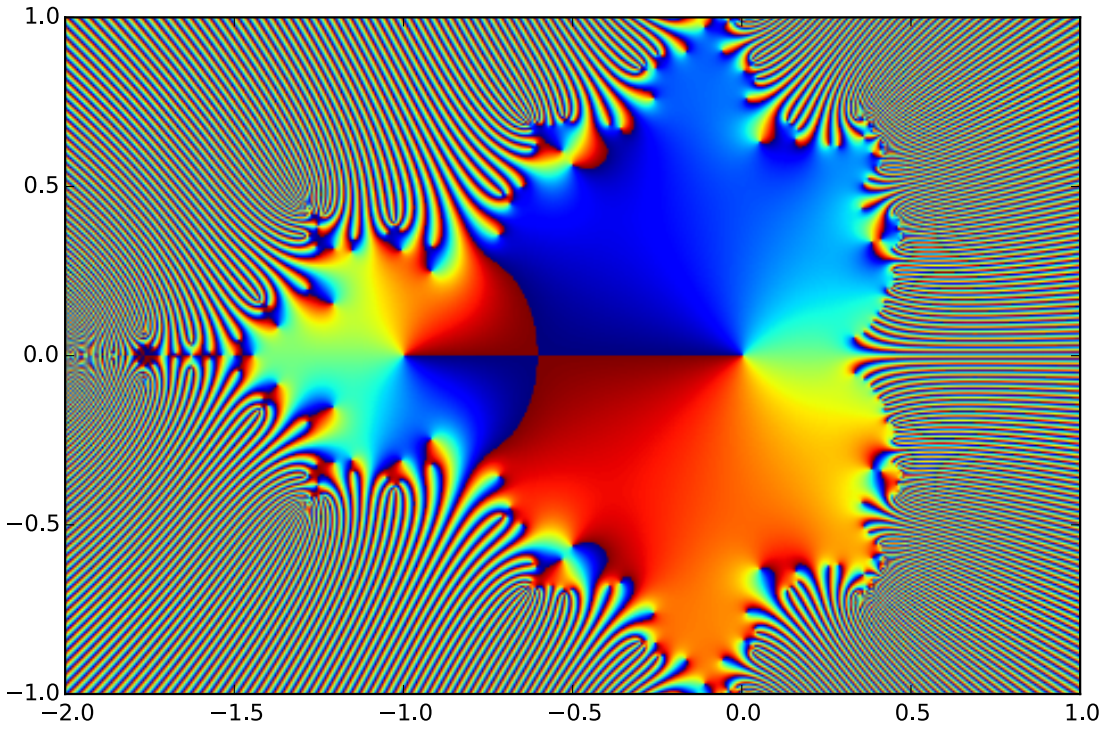
```
In [34]: (np.abs(z)<100.0)
```

```
Out[34]: array([[False, False, False, ..., False, False, False],
                [False, False, False, ..., False, False, False],
                [False, False, False, ..., False, False, False],
                ...,
                [False, False, False, ..., False, False, False],
                [False, False, False, ..., False, False, False],
                [False, False, False, ..., False, False, False]], dtype=bool)
```

```
In [35]: ax = plt.imshow(np.log(abs(z)), extent=(-2,1,-1,1));
         plt.colorbar(ax)
         plt.show()
```

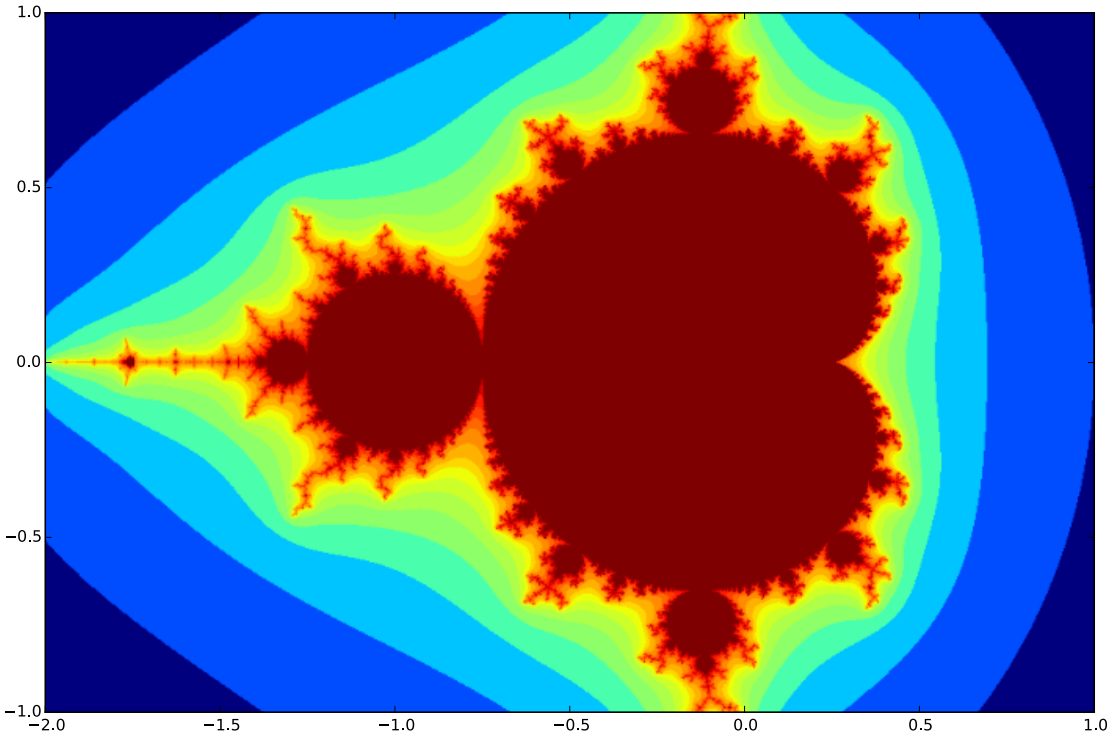


In [36]: plt.imshow(np.angle(z), extent=(-2,1,-1,1));

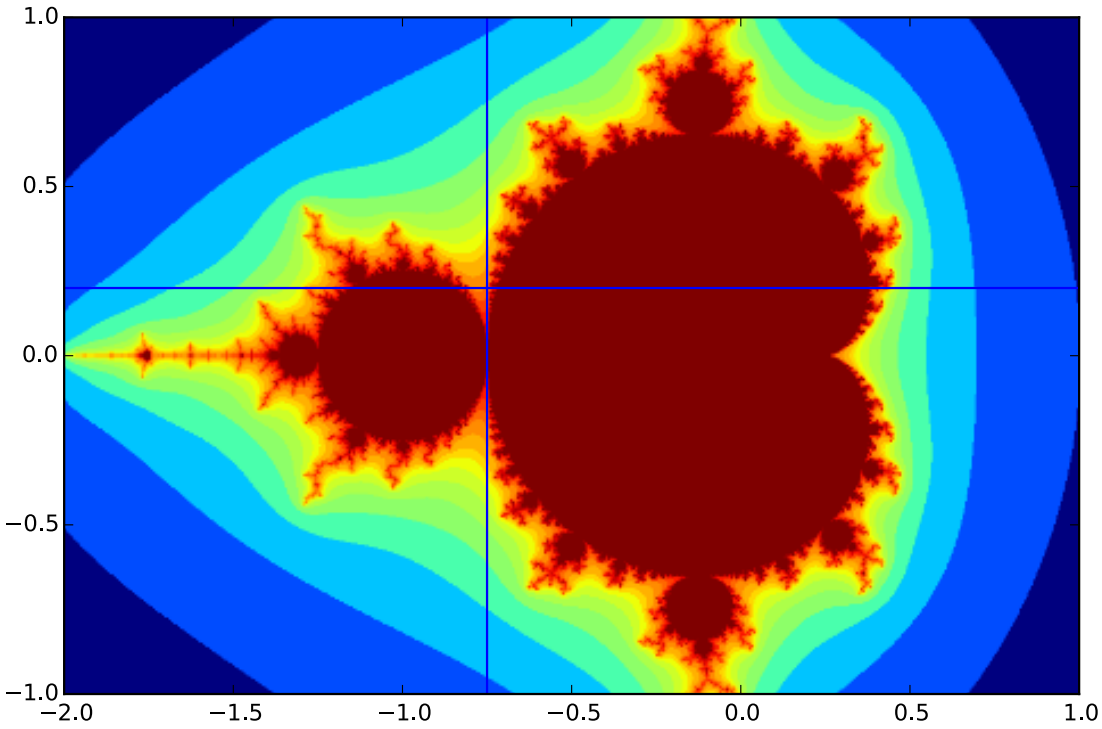



```
In [37]: def mandel2(c, numIter=10, maxabs=5):
             z = c
             num = np.zeros_like(c, dtype=np.int16)
             for i in range(numIter):
                 mask = (abs(z)<maxabs)
                 z = mask*(z**2 + c)
                 num += (abs(z)<maxabs)
             return z, num
```

```
In [38]: z, num = mandel2(c, 100, 5)
         plt.figure(figsize=(12,9))
         plt.imshow(num, extent=(-2,1,-1,1));
```




```
In [39]: plt.imshow(num, extent=(-2,1,-1,1));
         plt.axhline(0.2)
         plt.axvline(-0.75)
         plt.show()
```



1.6 Übung matplotlib

- Suche eine interessante Stelle im Apfelmännchen und markiere diese im Plot
- Suche neue Grenzen um diese Stelle herum, berechene neu und zeichne die Vergrößerung
- Bestimme und markiere einige Punkte in der Vergrößerung und plotte die zugehörigen Reihen

In [ ]:

1.7 4. SymPy

SymPy ist ein Computer Algebra System das in reinem Python geschrieben ist. * BSD Lizensiert, als Bibliothek in (kommerzieller) Software nutzbar * Leichtgewichtig (im Gegenastz zu Sage)

```
In [40]: import sympy as sp
         sp.init_printing()
         a, b, x, y = sp.symbols("a b x y")
         a, b, x, y
```

Out[40]:

$$(a, \quad b, \quad x, \quad y)$$

Die Variablen a, b, x und y sind nun symbolische Werte, mit denen man in Python-Syntax Ausdrücke und Gleichungen eingeben kann:

```
In [41]: greekSyms = sp.symbols("alpha beta gamma delta lamda Delta Sigma")
         greekSyms
```

Out[41]:

$$(\alpha,\quad \beta,\quad \gamma,\quad \delta,\quad \lambda,\quad \Delta,\quad \Sigma)$$

In [42]:
```
alpha, beta, gamma = greekSyms[:3]
```

In [43]:
```
f, g, h = sp.symbols("f g h", cls=sp.Function)
type(f)
```

Out[43]:
```
sympy.core.function.UndefinedFunction
```

In [44]:
```
a+b, sp.diff(f(alpha)), sp.sqrt(a**2+b/gamma), sp.Eq(x+y,gamma), sp.Sum(x**a, (a, 1, b)), sp.I
```

Out[44]:

$$\left(a+b,\quad \frac{d}{d\alpha}f(\alpha),\quad \sqrt{a^2+\frac{b}{\gamma}},\quad x+y=\gamma,\quad \sum_{a=1}^{b} x^a,\quad \int_1^\infty \frac{1}{x^2}\,dx\right)$$

Für Ausdrücke wie Integrale, die eventuell ausgewertet werden können, kann man mit der Methode `.doit()` diese Auswertung anstossen:

In [45]:
```
intExpr = sp.Integral((1/x)**2,(x,1,sp.oo))
sp.Eq(intExpr, intExpr.doit())
```

Out[45]:

$$\int_1^\infty \frac{1}{x^2}\,dx = 1$$

In [46]:
```
rootExpr = sp.sqrt(a**2+b/gamma)
sp.Eq(sp.Derivative(rootExpr, a), rootExpr.diff(a))
```

Out[46]:

$$\frac{\partial}{\partial a}\sqrt{a^2+\frac{b}{\gamma}} = \frac{a}{\sqrt{a^2+\frac{b}{\gamma}}}$$

Merke: Großgeschriebene Bezeichner (`Sum`, `Derivative`, `Integral`) erhalten die symbolische Operation, kleingeschriebene (`sum`, `diff`, `integrate`) Verben führen die Operation gleich durch.

In SymPy-Ausdrücken kann man mit `sp.subst` konkrete Werte einsetzen:

In [47]:
```
valDict = dict(a=sp.pi, b=sp.E)
(a+b).subs(valDict)
```

Out[47]:

$$e+\pi$$

Wer es noch konkreter mag, kann `evalf` verwenden:

In [48]:
```
(a+b).subs(valDict).evalf(100)
```

Out[48]:

5.859874482048838473822930854632165381954416493075065395941912220031893036639756593199417003867283495

In [49]:
```
binom = (a+b)**5
binom.doit()
```

Out[49]:

$$(a+b)^5$$

`doit()` führt nur Berechnungen aus (wie Summen, Ableitungen und Integrale), für Umformungen stehen unter anderem Folgende Funktionen zur Verfügung:

In [50]:
```
polynom = sp.expand(binom)
polynom
```

Out[50]:

$$a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5$$

In [51]:
```
sp.factor(polynom)
```

Out[51]:

$$(a+b)^5$$

In [52]:
```
sp.tan(x).rewrite(sp.sin)
```

Out[52]:

$$\frac{2\sin^2(x)}{\sin(2x)}$$

Gleichungen kann man mittels `solve` nach einer Variable auflösen:

In [53]:
```
equ = x**2+a*x+b
quad = sp.Eq(equ, 0)
sp.solve(quad, x)
```

Out[53]:

$$\left[-\frac{a}{2} - \frac{1}{2}\sqrt{a^2-4b},\quad -\frac{a}{2} + \frac{1}{2}\sqrt{a^2-4b}\right]$$

In [54]:
```
sp.sin(x).series(x,n=10) #.removeO()
```

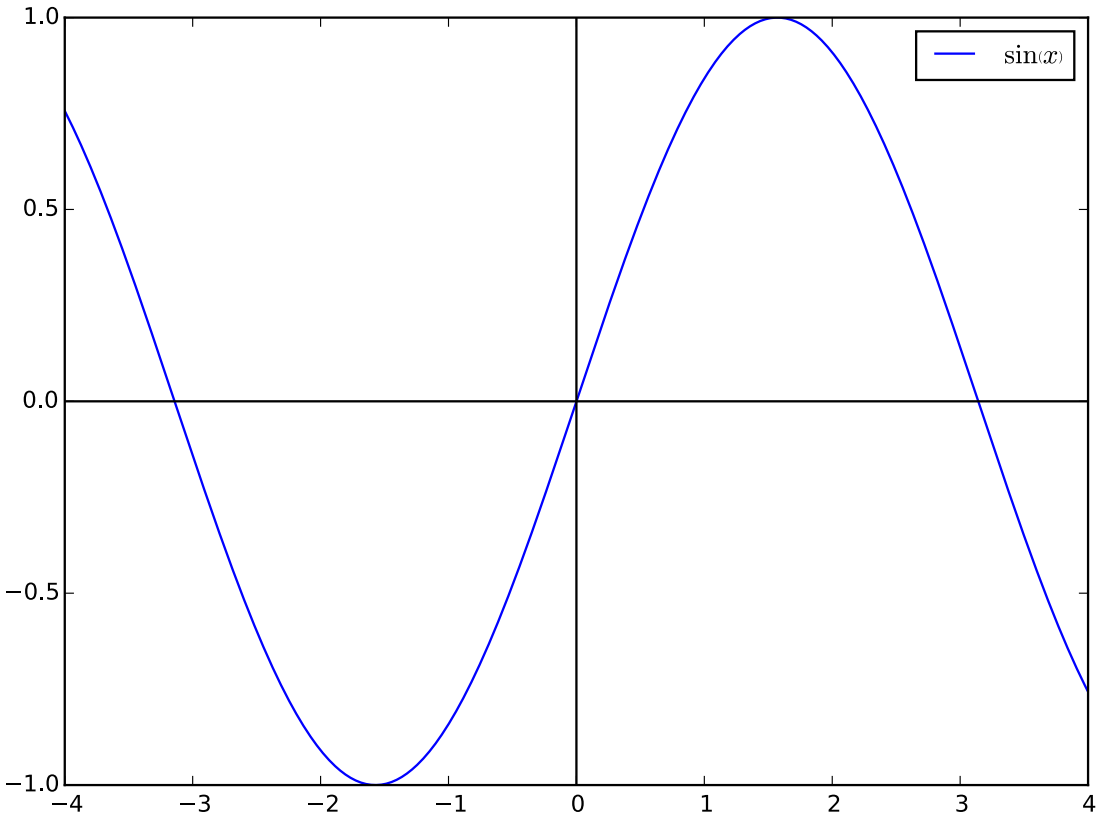
Out[54]:

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} + \mathcal{O}\left(x^{10}\right)$$

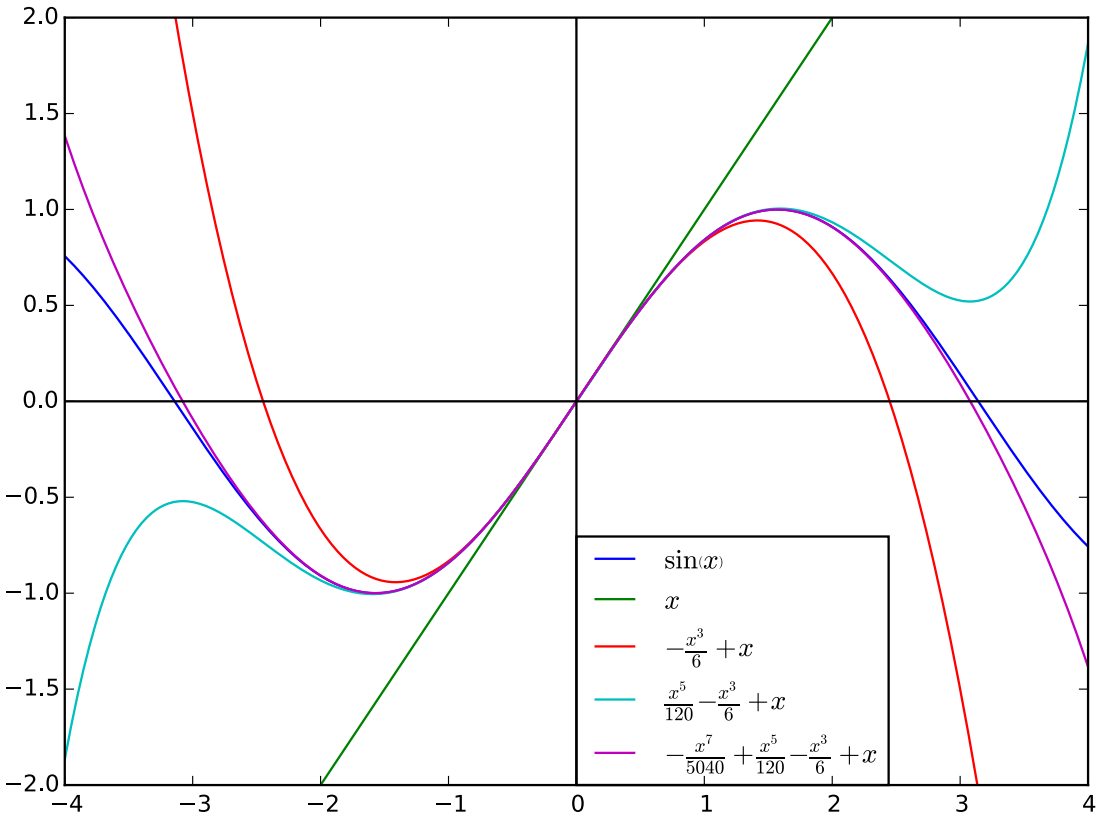
1.7.1 Zusammenspiel `sympy` und `numpy`

Symbolische Berechnungen sind schön und gut, aber auch recht langsam. die Funktion `lambdify` kann einen `sympy`-Ausdruck in eine `numpy`-Funktion umwandeln, die man dann wie gehabt plotten kann.

In [55]:
```
def toLaTeX(symExpr):
    return "$"+sp.latex(symExpr)+"$"

domain = np.linspace(-4, 4, 1000)
origFunc = sp.sin(x)
numFunc = sp.lambdify(x, origFunc, "numpy")
plt.plot(domain, numFunc(domain), label=toLaTeX(origFunc))
plt.ylim([-1.0, 1.0])
plt.axvline(x=0, color='black')
plt.axhline(y=0, color='black')
plt.legend()
plt.show()
```




```
In [56]: domain = np.linspace(-4, 4, 1000)
         origFunc = sp.sin(x)
         numFunc = sp.lambdify(x, origFunc, "numpy")
         plt.plot(domain, numFunc(domain), label=toLaTeX(origFunc))
         for i in range(11)[3::2]:
             symFunc = origFunc.series(x, n=i).removeO()
             numFunc = sp.lambdify(x, symFunc, "numpy")
             plt.plot(domain, numFunc(domain), label=toLaTeX(symFunc))
         plt.ylim([-2.0, 2.0])
         plt.axvline(x=0, color='black')
         plt.axhline(y=0, color='black')
         plt.legend(loc=(0.5,0))
         plt.show()
```



1.7.2 Übung Ableitungen von $\sin(x)$

Berechne die Abeitungen von $\sin(x)$ für $x \in \{-3, -2, -1, 0, 1, 2, 3\}$ und zeichne die Tangenten.

In [ ]:

1.8 Dateiformat von Notebooks

Die Notebooks werden im im JSON-Format in Dateien mit der Endung `.ipynb` gespeichert.

Diese lassen sich per

> `ipython nbconvert --to latex|pdf|html` Dateiname.ipynb

nach LaTeX, PDF, HTML und weitere Formate exportieren.

Allerdings wird dafür eine funktionierende LaTeX-Installation sowie pandoc vorausgesetzt.

Bilder und Formeln werden in einer String-Codierung gespeichert, wodurch Notebook-Files schnell anwachsen können. Um die Dateien klein zu halten und reproduzierbare Ergebnisse zu erhalten, kann man vor dem Speichern über `Cell -> All Output -> clear` die Ausgaben löschen und nach dem Laden `Cell -> Run All` ausführen.