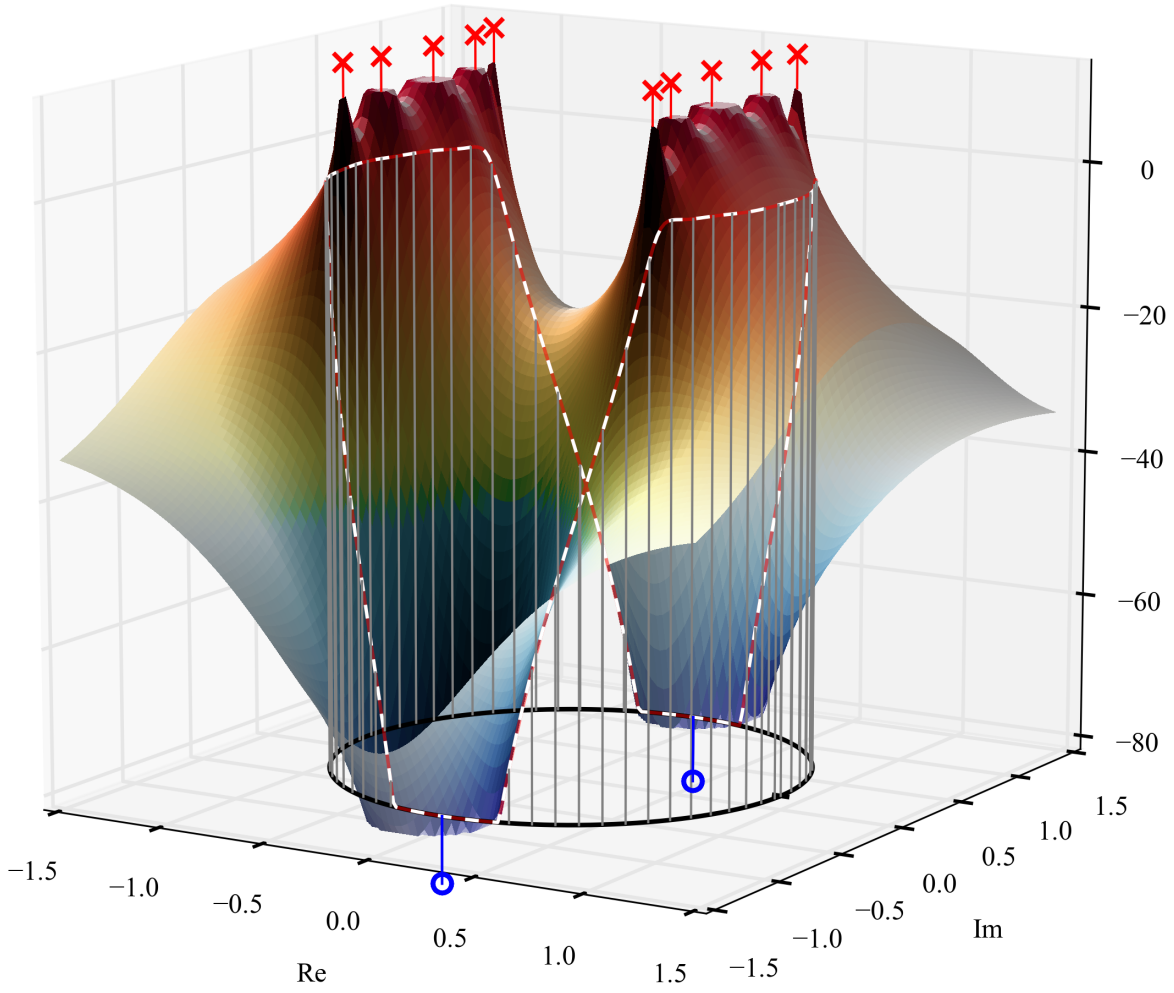
Digitale Signalverarbeitung mit Python

NumPy, SciPy und Matplotlib

Dr. Christian Münker



10. Februar 2017

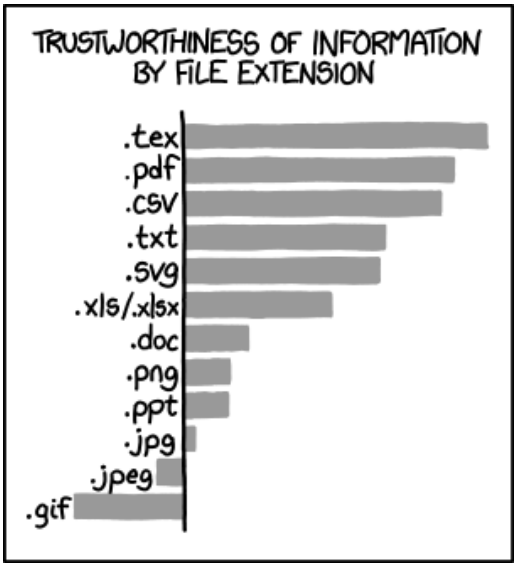
Chipmuenk@gmail.com

Digitale Signalverarbeitung mit Python: Numpy, Scipy und Matplotlib

Die Titelgrafik wurde erzeugt mit pyFDA (https://github.com/chipmuenk/pyFDA).

Überarbeitete Version vom 10. Februar 2017.

Diese Unterlagen wurden mit LaTeXverfasst, weil ...



Das Fileformat Ihres Vertrauens [(c) Randall Munroe, http://xkcd.com/1301/]

Inhaltsverzeichnis

1. Och nö ...

Och nöö ...

... noch eine Skriptsprache? Muss das denn sein? Reichen denn nicht schon Matlab, Javascript, Perl, Ruby, ... ? Genau das war meine Reaktion als mir Anfang 2012 ein Werkstudent Python als Ersatz für Matlab™ vorschlug. Zu diesem Zeitpunkt hatte ich bereits ein gutes Jahr nach Open Source Alternativen zu Matlab™ gesucht, um den Lizenzproblemen beim Einsatz in der Lehre und vor allem bei Kooperationen mit mittelständischen Firmen zu entkommen.

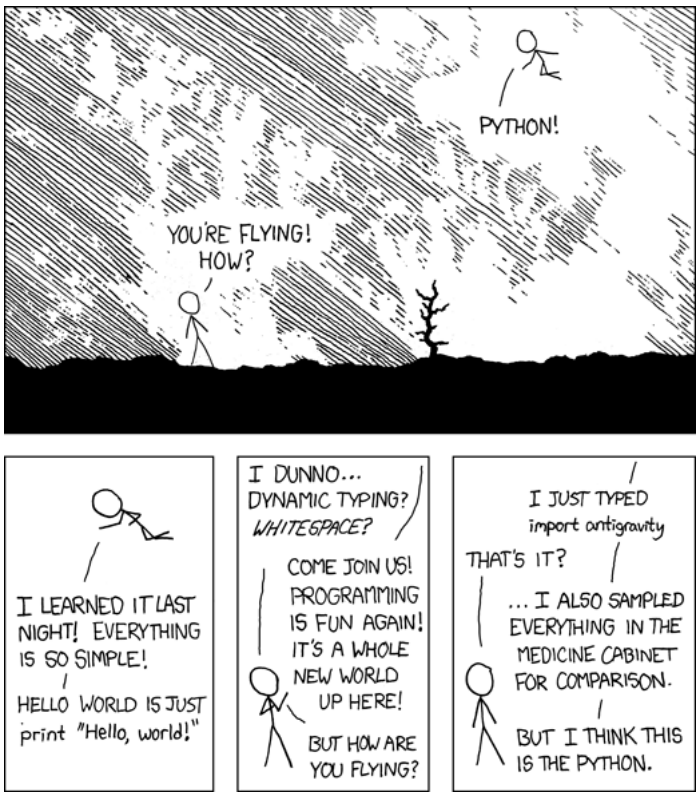


Abb. 1.1.: Python hebt ab! [(c) Randall Munroe, http://xkcd.com/353/]

In einem ersten Selbstversuch brauchte ich dann ca. 8 Stunden, um ohne Vorkenntnisse ein Matlab™-Skript zum digitalen Filterentwurf mit ca. 200 Zeilen Code erfolgreich nach Python zu „übersetzen“. Das überzeugte mich, zumal die Qualität der erzeugten Grafiken locker mit denen von Matlab™ mithalten konnte (Abb. 1.2).

Python wurde auf http://bit.ly/1jBjyWI im Januar 2014 als „Beste Programmiersprache für Einsteiger“ gekrönt. Als Hauptgründe wurden genannt:

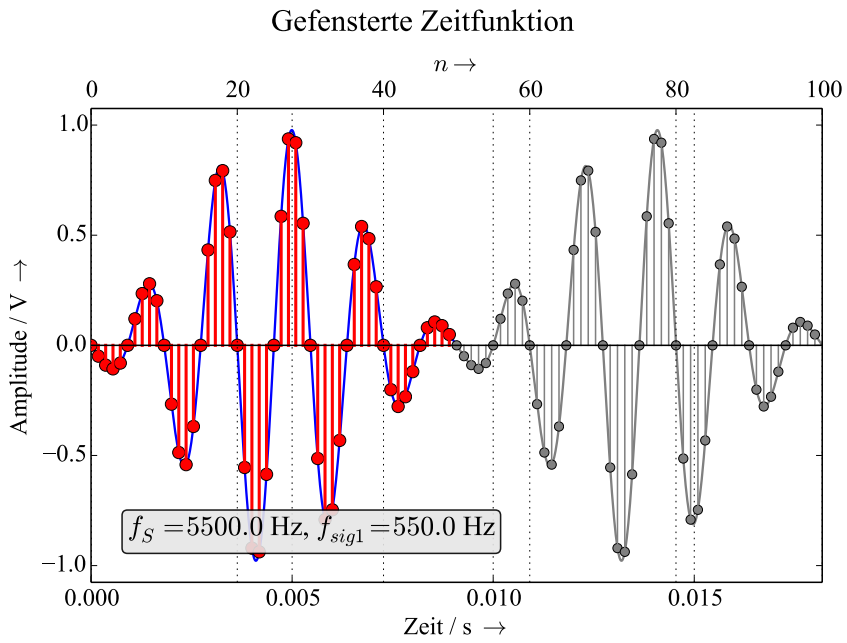


Abb. 1.2.: Beispiel für Python-Plot (Matplotlib)

Lesbarkeit: Man wird es entweder lieben oder hassen, Code durch Einrückungen anstelle geschweifter Klammern zu strukturiere; es hilft aber vor allem Anfängern lesbaren Code zu schreiben. Jedenfalls besseren als z.B. mit Perl, dessen Code manchmal aussieht als ob er von einer fluchenden Comicfigur stammt ...

Universalität: ... Python ist im Gegensatz zu Matlab™ eine „richtige“ Programmiersprache, mit mächtigen Funktionen für Datenbank- und webbasierte Operationen, I/O - Funktionen, Software-Test, professionelle GUI-Erstellung etc. Es ist damit auch für größere Projekte geeignet. Matlab™ merkt man an einigen Stellen an, dass es über mehr als 20 Jahre gewachsen ist und nicht auf einer „richtigen“ Programmiersprache basiert - der Blog **http://abandonmatlab.wordpress.com** ist eine polemische aber unterhaltsame Auseinandersetzung mit den Schwächen von Matlab™.

Batteries included: Sehr viele Funktionen werden bereits von den mitgelieferten Python Standard Libraries abgedeckt (unbedingt anschauen: http://docs.python.org/2/library/) oder lassen sich leicht mit einem zusätzlichen Modul nachrüsten.

Multi-Paradigm: Python ist eine objektorientierte Sprache, die Objektorientiertheit aber nicht erzwingt. Prozedurale oder funktionale Programmierung sind genauso möglich.

Cross-Plattform: Python und GUI-Bibliotheken wurden auf verschiedenste Plattformen portiert. Neben den „großen drei“ (Windows, Linux, OS X) werden u.a. auch Android und iOS (Kivy) und Raspberry Pi unterstützt.

Support: Bei aktiven Open Source Projekten gibt es meist sehr gute und rasche Unterstützung durch die Community, mindestens genauso gut und eher schneller als bei kommerziellen Produkten.

Web-Based Computing: Mit IPython kann man interaktive Notebooks erstellen, vergleichbar mit den Worksheets von MathCAD, Mathematica oder Maple. Diese Notebooks können als Files weitergegeben oder auf einem Server bereitgestellt und dann im Browser bearbeitet werden (Webbased Computing). Für die Lehre ergeben sich damit völlig neue Möglichkeiten.

Aus ähnlichen Gründen ist Python die Sprache #1 für die Programmierausbildung an amerikanischen Universitäten laut http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext (2014). Ende 2015 überholte „learn python“ in Google Trends „learn java“ (https://dzone.com/articles/learn-python-overtakes-learn-java).

1.1. Wer nutzt es?

Python ist weit verbreitet, laut TIOBE-Index (http://tiobe.com) war es die Programmiersprache der Jahre 2007 und 2010 mit der höchsten Zuwachsrate an Nutzern, sie steht 2012 auf Platz 8 der Beliebtheitsskala, 2015 auf Platz 5. Interessant ist es auch einen Blick in die Jobangebote der Python Gruppen von Xing und LinkedIn zu werfen. Laut http://blog.codeeval.com/codeevalblog/2015 ist Python seit 4 Jahren die bei Arbeitgebern am meisten gefragte Programmiersprache.

Google setzt sehr stark auf Python und unterstützt die Community – viele Pakete, Programme und auch die python(x,y) Distribution werden von Google gehostet. Gründe hierfür werden die Universalität der Sprache sein für den Einsatz als „Glue Software“ zwischen verschiedenen Anwendungen sowie die Verfügbarkeit der Module für Web Applikationen und „Big Data“.

Libre Office ersetzt Java nach und nach durch Python als Makro-Sprache (http://wiki.openoffice.org/wiki/Python_as_a_macro_language).

Web Frameworks sind eins der Haupteinsatzgebiete von Python - siehe http://wiki.python.org/moin/WebFrameworks. Die bekanntesten Frameworks sind vermutlich Django und Zope (z.B. verwendet für das CMS Plone).

Ausbildung: Eine Vielzahl von weiterführenden Informationen und Links zum Einsatz von Python an Universitäten und in der Ausbildung bietet http://www.python.org/community/sigs/current/edu-sig/.

Test and Measurement: In vielen Firmen (z.B. Rohde & Schwarz, Bosch) wird Python zur Messautomatisierung in Labor und Fertigung eingesetzt.

Spielzeug: Python taugt auch, um mit populärer Hardware zu reden: So wurde Python von der **Raspberry Pi** Stiftung als die offizielle Lehrsprache für den Pi ausgewählt: http://raspberrycenter.de/handbuch/python-programmierung, **Arduino-Boards** können mit http://playground.arduino.cc/interfacing/python über Python angesprochen werden. **Lego Mindstorms** und **Fischertechnik** lassen sich ebenfalls steuern, siehe z.B. http://digitalcommons.bryant.edu/cgi/viewcontent.cgi?article=1017&context=sci_jou bzw. http://py4ft.weebly.com.

Applikationen haben oft ein Herz aus Python, Beispiele hierfür sind

- der File-Hosting Service **dropbox** (http://dropbox.com),
- der eBook-Converter **Calibre** (http://calibre-ebook.com),
- der praktische ToDo - Manager **Task Coach** (http://www.taskcoach.org),
- der Cross-Platform Text und Source Code Editor **Sublime Text** (http://sublimetext.com - unbedingt anschauen, wer ihn noch nicht kennt!).
- der ursprüngliche BitTorrent Client **BitTorrent** (http://bittorrent.com)
- das 3D Modellierungs- und Animationsprogramm **Blender 3D** (http://www.blender.org) mit Python Skripting Engine

1.2. Warum ist das so GUT?!

1.2.1. Scientific Computing

In allen Zweigen der Natur- und Ingenieurwissenschaften werden „Computational Science“ Softwareprodukte intensiv genutzt, u.a. aus den folgenden Gründen:

Interaktiver Interpreter ermöglicht „exploratorisches Arbeiten“ (= Herumspielen) und spielerisches Lernen sowie schnelle Debugzyklen. Im Zeitalter von GHz-Prozessoren ist auch die vergleichsweise geringe Ausführungsgeschwindigkeit von interpretierten Sprachen nur noch in wenigen Fällen ein Problem, meist ist „langsam“ = „schnell genug“.

Hochsprache: Mächtige Befehle führen zu kompakten Programmen und kurzen Entwicklungszeiten.

Numerische Arrays: Leistungsfähige Array-Syntax ermöglicht einfaches und effizientes Arbeiten mit numerischen Arrays (Listen, Vektoren, Matrizen, Tensoren). Der vektorisierte Code ohne For-Next Schleifen ist schnell (hochoptimierte Libraries für lineare Algebra) und (meist) übersichtlich.

Mächtige Plotting Tools stellen Information übersichtlich dar, geben schnelles Feedback und erzeugen Grafiken in vielfältigen Formaten für Publikationen und Präsentationen.

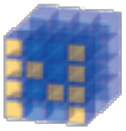
Matlab™ / Simulink sind dabei ohne Zweifel (noch) am weitesten verbreitet, aber Python erfüllt diese Anforderungen genauso gut und ist bereits die Open Source Sprache #1 für Scientific Computing (siehe Kap. 1.2.2). Auch für die (digitale) Signalverarbeitung gibt es kaum Aufgaben, die man nicht auch mit Python erledigen könnte:

1.2.2. Scientific Python

Einen Überblick über Aktivitäten rund um „Scientific Python“ (Scipy) bieten die Site des gleichnamigen Pythonmoduls http://scipy.org und der zugehörigen Konferenz http://conference.scipy.org (dreimal jährlich - jeweils einmal in Amerika, Europa und Indien) und der Blog **Planet Scipy** http://www.swc.scipy.org/.

Thematisch noch vielfältiger sind die allgemeinen Python Konferenzen **PyCon** https://us.pycon.org bzw. https://de.pycon.org/. Ein Blick auf die Sponsoren zeigt, dass es hier nicht (nur) um akademische Beweihräucherung geht.

Die Entwickler haben einen „Scipy Stack“ http://www.scipy.org/stackspec.html an Tools und Libraries definiert als Grundvoraussetzung für das wissenschaftliche Arbeiten mit Python (siehe Tab. 1.1). http://scipy-lectures.github.io/ gibt eine knappe aber gute Einführung in die wichtigsten Elemente von Python und der wissenschaftlichen Module.

	**Python**	Der eigentliche Python-Intpreter und die mitgelieferten Standardlibraries
	**numpy**	(Numeric Python) ist u.a. eine Bibliothek für schnelle Vektor- und Matrixrechnungen mit ähnlicher Funktionalität wie Matlab™ (ohne Grafik und Toolboxen)
	**matplotlib**	ist die passende leistungsfähige 2D - Grafikbibliothek mit grundlegenden 3D - Funktionen
	**scipy**	(Scientific Python) bietet Signalverarbeitung, Statistik, Regelungstechnik, spezielle Funktionen ... hier findet man die Funktionalität von vielen Matlab™-Toolboxen wieder.
	**IPython**	(Interactive Python) stellt eine komfortable interaktive Shell und ein browserbasiertes Notebook zur Verfügung zum Entwickeln und Debuggen von Code und für webbased Computing.
	**sympy**	ist eine Bibliothek für symbolische Mathematik (vergleichbar der Symbolic Math Toolbox von Matlab™)
	**pandas**	ist ein Modul zur Datenanalyse

Tab. 1.1.: Die wichtigsten Bestandteile des Scientific Python Stack

Einen Einblick in zwei der wichtigsten Werkzeuge für Scientific Computing - die IDE **Spyder** und interaktive Console / Notebook **IPython** gibt das Video http://pyvideo.org/video/2113/the-advantages-of-a-scientific-ide-scipy-2013-pr-1. Einen Eindruck von der Vielfalt der Python-basierten wissenschaftlichen Bibliotheken und Anwendungen gibt http://www.scipy.org/topical-software.html.

1.2.3. Python als Programmiersprache

Python wurde auf http://bit.ly/1jBjyWI im Januar 2014 als „Beste Programmiersprache für Einsteiger“ gekrönt. Als Hauptgründe wurden genannt:

Lesbarkeit: Man wird es entweder lieben oder hassen, Code durch Einrückungen anstelle geschweifter Klammern zu strukturiere; es hilft aber vor allem Anfängern lesbaren Code zu schreiben. Jedenfalls besseren als z.B. mit Perl, dessen Code manchmal aussieht als ob er von einer fluchenden Comicfigur stammt ...

Universalität: ... Python ist im Gegensatz zu Matlab™ eine „richtige“ Programmiersprache, mit mächtigen Funktionen für Datenbank- und webbasierte Operationen, I/O - Funktionen, Software-Test, professionelle GUI-Erstellung etc. Es ist damit auch für größere Projekte geeignet. Matlab™ merkt man an einigen Stellen an, dass es über mehr als 20 Jahre gewachsen ist und nicht auf einer „richtigen“ Programmiersprache basiert - der Blog **http://abandonmatlab.wordpress.com** ist eine polemische aber unterhaltsame Auseinandersetzung mit den Schwächen von Matlab™.

Batteries included: Sehr viele Funktionen werden bereits von den mitgelieferten Python Standard Libraries abgedeckt (unbedingt anschauen: http://docs.python.org/2/library/) oder lassen sich leicht mit einem zusätzlichen Modul nachrüsten.

Multi-Paradigm: Python ist eine objektorientierte Sprache, die Objektorientiertheit aber nicht erzwingt. Prozedurale oder funktionale Programmierung sind genauso möglich.

Standalone Applikationen: Dank leistungsfähiger GUI-Bibliotheken wie Qt, Kivy oder GTK können Applikationen erstellt werden, die man mit relativ geringem Aufwand in Standalone-Executables umwandeln kann (s. Kapitel B). Im Gegensatz zu Matlab™, Labview oder Mathematica muss man keine mehrere 100 MB große Library mitliefern.

Cross-Plattform: Python und GUI-Bibliotheken wurden auf verschiedenste Plattformen portiert. Nicht nur die „großen drei“ (Windows, Linux, OS X) werden unterstützt, sondern auch Android und iOS (Kivy) und Exoten wie Raspberry Pi.

Support: Bei aktiven Open Source Projekten gibt es meist sehr gute und rasche Unterstützung durch die Community, mindestens genauso gut und eher schneller als bei kommerziellen Produkten.

Web-Based Computing: Mit IPython kann man interaktive Notebooks erstellen, vergleichbar mit den Worksheets von MathCAD, Mathematica oder Maple. Diese Notebooks können als Files weitergegeben oder auf einem Server bereitgestellt und dann im Browser bearbeitet werden (Webbased Computing). Für die Lehre ergeben sich damit völlig neue Möglichkeiten.

Einen flockig geschriebenen Überblick bietet „Python Programming Language Advantages“ von Mike Levin (http://mikelev.in/2011/01/python-programming-language-advantages/, Jan. 2011), das Python bewusst nicht mit anderen Programmiersprachen vergleicht, sondern nur die Vorzüge schildert.

Und es gibt immer mehr Umsteiger: Laut einer Studie des ACM vom Juli 2014 ist Python die Sprache #1 für Einführungskurse in Computerwissenschaften an den Top-US Universitäten (http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities) und hat Java verdrängt.

University of Oslo, „Experience with Using Python as a Primary Language for Teaching Scientific Computing at the University of Oslo“, 2012, http://hplgit.github.com/edu/uiopy/uiopy.html. Der Bericht schildert wie über mehrere Jahre hinweg Python in die Mathematik-, Software- und Physikausbildung eingeführt wurde und dort Maple, Matlab™ und Java verdrängt hat. Ein schönes Beispiel von interdisziplinärer Zusammenarbeit ...

Vincent Noel, Europython 2011, „Python(x,y): Python as your next MATLAB“ (http://www.lmd.polytechnique.fr/~noel/EP2011/PythonAsYourNextMatlab.zip) und „Diving into scientific Python“ (http://www.lmd.polytechnique.fr/~noel/EP2011/DivingIntoScientificPython.zip), Tutorials für Anwender mit Vorkenntnissen in Matlab™ bzw. Python.

Don MacMillen, „From MATLAB Guide to IPython Notebook“, April 2014, http://blogs.siam.org/from-matlab-guide-to-ipython-notebook/ hat das erste Kapitel von D.J. Higham and N.J. Higham's Buch „MATLAB Guide“ in ein interaktives IPython Notebook umgesetzt, das an praktischen (und interessanten) Beispielen zeigt, was (I)Python kann, nebenbei bekommt man den Einstieg in Python beigebracht.

Auszüge aus dem Thread „Python vs. Matlab“, Mai 2013 in der LinkedIn Gruppe „Scientific Python“

> „[...] Matlab is not the tool of choice for large software projects.
>
> Because of the large number of people developing in Python we will continue to see an improvement in tools useful for scientific computation. Python can already compete directly with Matlab in terms of computational tools, however, to make best use of these one needs to develop deeper software writing skills – skills that result in easier to maintain and better documented code.
>
> Building and maintaining a productive engineering design department based on either Python or Matlab involves significant up front investment (building software carpentry skills vs. software licencing costs), however, after the institutional knowledge builds up I think you will find Python to be a less expensive option in the long term.
>
> One should also consider the future. Python is increasingly being used to create useful and open source tools for scientists and engineers (with excellent documentation and easy to maintain code). Tools that have yet to be created should be a factor in the decision – as should the people creating them. How many kids are learning Python using their Raspberry Pi computers? How many of these are also learning Matlab? Will web-based computation become the norm in the future? From this point of view, how easy is it to build interactive websites with Matlab?
>
> Using Python ensures you are investing in the future.“

David Schryer

1.2.4. Lizenzen und Rechtliches

Die finanziellen Nachteile von kommerzieller Software im Vergleich zu Free, Open-Source Software (FOSS) liegen auf der Hand, es gibt aber noch weitere, nicht so offensichtliche Probleme:

Rechtssicherheit: Unterschiedlichen Lizenztypen und deren unterschiedlichem Funktionsumfang und Benutzungsrechten führen zu einer unübersichtlichen Situation, manche Studierenden, Dozenten und Mitarbeiter bewegen sich absichtlich oder unabsichtlich jenseits der Legalität: Studierende suchen sich „alternative Beschaffungswege“, weil sie keine Lizenz bekommen oder ihnen das Prozedere zu umständlich ist. Doktoranden / Dozenten in Forschungs- und Industrieprojekten haben nicht immer die erforderliche Lizenz.

Zukunftssicherheit: Speziell bei Programmiersprachen, die nur von einem Anbieter lizensiert werden wie Matlab^{TM} (im Gegensatz zu z.B. VHDL oder C), ist man dem Anbieter völlig ausgeliefert. Änderungen der Lizenzpolitik (Mathworks spaltet z.B. Toolboxen immer weiter auf, so dass mehr Toolboxen gekauft werden müssen) oder der Finanzsituation (Studienbeiträge!) können dazu führen, dass die Software plötzlich nicht mehr im gewünschten Umfang zur Verfügung steht.

Mitmachen: Studierende können sich aktiv an Open Source Projekten beteiligen -> gute Motivation!

Weitergabe von Code: Bei Kooperationen mit mittelständischen Industriepartnern kann nicht vorausgesetzt werden, dass auch dort Lizenzen für Matlab^{TM} mit allen Toolboxen in ausreichender Zahl zur Verfügung stehen. In Drittmittelprojekten müssen daher u.U. zunächst Volllizenzen an der HM, danach noch einmal beim Industriepartner erworben werden.

1.2.5. Und Simulink?

Nein, es gibt in Python noch keinen universellen Ersatz für Simulink: Simulink ist ein grafisches “Signal-Flow Tool”, mit dem sich Systeme der Regelungstechnik und der Signalverarbeitung modellieren und simulieren lassen. Eine Stärke von Simulink ist die automatische Codegenerierung für DSPs, FPGAs und Embedded Prozessoren.

Für verschiedene Teilbereiche gibt es jedoch grafische Simulationstools auch in Python:

GnuRadio / GnuRadioCompanion (GRC) (http://gnuradio.org/) kann graphisch Signalfluss-Diagramme der digitalen Signalverarbeitung erstellen und in Real-Time simulieren. Der geplante Einsatzbereich ist Software Defined Radio, es gibt daher verschiedene Hardware-Interfaces. Die einzelnen Blöcke bestehen aus kompiliertem C++ - Code und sind daher sehr schnell, Python wird als Scheduler und zum Verknüpfen der Blöcke eingesetzt. Details zum graphischen „Companion“ GRC finden sich unter http://www.joshknows.com/grc, ein Tutorial unter http://gnuradio.org/redmine/projects/gnuradio/wiki/TutorialsWritePythonApplications. Zum einfachen Ausprobieren kann man DVD-Images herunterladen (basierend auf Ubuntu).

OpenModelica (https://openmodelica.org/) ist eine OpenSource Simulations- und Entwicklungsumgebung für Modelica - Modelle mit Python-Interface OMPython. Modelica wird vor allem für Model Based Design in der Automobil-, Luftfahrt- und Kraftwerk-Industrie eingesetzt (https://www.modelica.org/).

DEVSimPy (http://code.google.com/p/devsimpy/) ist ein GUI zur Simulation von eventbasierten Systemen, das auf SimPy (https://simpy.readthedocs.org/, Kap. 9.2) aufsetzt.

1.3. Literatur und Kurse zu Python

Es gibt zahllose Kurse, Foliensätze, PDFs, Videos ... im Netz rund um Python; hier ist daher nur eine höchstwillkürliche Auswahl von mir. Einen allerersten Überblick gibt Wikipedia unter http://de.wikipedia.org/wiki/Python_%28Programmiersprache%29. Falls nicht ausdrückliche erwähnt, sind alle Angebote auf englisch. Eine größere Auswahl finden Sie unter http://olimex.wordpress.com/2014/06/12/collection-of-51-free-e-books-for-python-programming/

Und wenn Sie möglichst schnell durchstarten wollen: Der Kurs unter http://scipy-lectures.github.io/ gibt nur eine ganz kurze Einführung in Python und steigt dann direkt in typische wissenschaftliche Problemstellungen mit NumPy, Matplotlib, SciPy, SymPy u.a. ein. Mit vielen Beispielen und Übungsaufgaben!

1.3.1. Python als Programmiersprache

Interaktiv

Computer Science Circles, *„Grundlagen des Programmierens mit Python“*, interaktiver Python - Kurs auf deutsch für absolute Anfänger, http://cscircles.cemc.uwaterloo.ca/0-de/. Dieser Kurs wird zur Verfügung gestellt von der Initiative Bundeswettbewerb Informatik (www.bwinf.de). Nutzt das geniale **Visualisierungs-Tool** Pythontutor.com (unter „Visualisierung“):

Online Python Tutor, http://www.pythontutor.com/ ist ein freies Tool, das Schülern / Studierenden dabei hilft, Programmieren zu lernen: Beliebiger Python Code wird im Browser schrittweise abgearbeitet, dabei wird gezeigt wie sich Variablen und Objekte ändern.

Details siehe Philip J. Guo, „Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education“, in Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE), March 2013.

http://runestoneinteractive.org, *Interactive Python*, http://interactivepython.org/runestone/static/thinkcspy/index.html. Interaktiver Kurs mit Code Execution im Browser (basierend auf Skulpt, einer Javascript - Python - Implementierung)

Online

Software Carpentry, http://software-carpentry.org/v4/index.html, hat Python als Sprache gewählt, um Programmiermethoden und -konzepte zu unterrichten. Die Kurse bestehen aus fünf- bis zehnminütigen Videohäppchen und Folien. Der Kurs *„Python“* selbst gibt eine gute Einführung in die Sprache, empfehlenswert sind auch *„Object-Oriented Programming“*, *„Sets and Dictionaries“*, *„Testing“* und *„Version Control“*.

Bernd Klein, *„Python-Tutorial“*, http://www.python-kurs.eu/kurs.php bzw. http://www.python-kurs.eu/kurs.php für Python 3 ist ein sehr guter Kurs für Einsteiger und Fortgeschrittene auf Deutsch.

John B. Schneider (Digilent), *„Introduction to Algorithmic Problem Solving using Python “*, http://www.digilentinc.com/Classroom/IntroProgramming1/. Sehr ausführlicher Videokurs, der insgesamt mehrere Stunden lang die Grundlagen der Programmierung mit Hilfe von Python erklärt.

Zed. A. Shaw, *„Learn Python the Hard Way“*, sehr guter und ausführlicher Kurs für Anfänger. Als HTML-Code frei verfügbar unter http://learnpythonthehardway.org/, PDF und Video-Kurs für 29 $ erhältlich. „Hard Way“ bezieht sich übrigens darauf, dass man die Übungen *selber* lösen soll: Die Lektionen bestehen aus 52 Übungen, die anfangend von „Hello World“ zunehmend schwieriger werden bis hin zur letzten Übung, bei der die Engine für einfaches Web-basiertes Spiel geschrieben werden soll.

Allen B. Downey, *Think DSP - Digital Signal Processing in Python*, 159 S., Green Tea Press, Needham, Massachusetts, 2015. Frei downloadbar (CC Lizenz) unter http://greenteapress.com/thinkdsp/thinkdsp.pdf.

Allen B. Downey, *Think Python - How to Think Like a Computer Scientist*, 244 S., Green Tea Press, Needham, Massachusetts, 2015. Frei downloadbar (CC Lizenz) unter http://greenteapress.com/thinkpython2/thinkpython2.pdf.

python.org, *„Audio/Video Instructional Materials for Python“*, http://www.python.org/doc/av/. Eine Sammlung von Tutorial- und Konferenzvideos sowie Podcasts.

Josh Cogliati, *„Non-Programmers Tutorial For Python 2.6“*, 2005, http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python. Sehr schön erklärtes Wikibook für die ersten Schritte zur Python Programmierung, auch für Python 3: http://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python_3

Swaroop C. H., *„A Byte of Python“*, http://www.swaroopch.com/notes/Python. Ein sehr ausführliches Tutorial für Anfänger, das auch als PDF erhältlich ist (Python 3).

Khan Academy, *„Introduction to Computer Science / Python Programming“* https://www.khanacademy.org/science/computer-science, kurzer und gut verständlicher Einsteigerkurs.

Peter Kaiser und Johannes Ernesti, *„Python - Das umfassende Handbuch“*, Galileo Computing, 819 S., 2008. Diese (veraltete, bezieht sich auf Python 2.5) Ausgabe ist vollständig online verfügbar unter http://openbook.galileocomputing.de/python/index.htm. Das deutschsprachige Buch bietet einen guten allgemeinen Einstieg in Python und geht auch auf fortgeschrittene Themen wie Qt, Web-Programmierung mit Django oder Netzwerkkommunikation ein. Eine aktuelle Version (2012) ist in Papier- und elektronischer Form (käuflich) erhältlich.

Mark Pilgrim, *Dive Into Python*, frei erhältlich unter CC-Lizenz unter http://www.diveintopython.net (2004) bzw. http://www.diveintopython3.net (2011) als HTML und PDF. Sehr gut erklärt.

python.org, *„The Python-Tutorial“*, http://docs.python.org/2.7/tutorial/. Das „offizielle“ Python - Tutorial, gut erklärt, aber eher für Anwender mit etwas Erfahrung.

Papier

Michael Weigend, *Raspberry Pi programmieren mit Python*, mitp-Verlag, 1. Auflage 2013, 448 Seiten, 29,95 €. Wer sich eh mit dem RPi beschäftigen wollte, findet mit diesem Buch eine gute Mischung von Python-Anleitung und kleinen Raspberry-Pi Projekten.

Wer tiefer in Python einsteigen möchte, wird vom folgenden Buch nicht enttäuscht werden:

—, *Python 3: Lernen und professionell anwenden*, mitp-Verlag, 5. Auflage 2013, 808 S., 39,95 € vom gleichen Author.

> Insgesamt liegt der Schwerpunkt auf der praktischen Arbeit mit Python. Ziel ist es, die wesentlichen Techniken und dahinter stehenden Ideen anhand zahlreicher anschaulicher Beispiele verständlich zu machen. Zu typischen Problemstellungen werden Schritt für Schritt Lösungen erarbeitet. So erlernen Sie praxisorientiert die Programmentwicklung mit Python und die Anwendung von Konzepten der objektorientierten Programmierung.

M. Summerfield, *Rapid GUI Programming with Python and Qt*, 1. ed., Prentice Hall PTR, 2007, http://www.qtrac.eu/pyqtbook.html, hat zwar den Schwerpunkt PyQt, bietet aber auch eine allgemeine Schnelleinführung in Python, die vor allem für Programmierer mit Kenntnissen in anderen Sprachen geeignet ist.

Leider gibt es keine aktuellere Auflage, aber das Buch ist didaktisch einfach super. Eine frühere Version (?) ist Online verfügbar unter http://www.cs.washington.edu/research/projects/urbansim/books/pyqt-book.pdf

Allen B. Downey, *Think Python - How to Think Like a Computer Scientist*, 2015, gibt es unter http://thinkpython.com als kostenloses PDF (englisch) oder als deutsches Buch „Programmieren Lernen mit Python“ für 25 € beim O'Reilly Verlag.

RRZN, *Python*, ist ein Skript vom Regionalen Rechenzentrum für Niedersachsen, das es für kleines Geld ($<$ 5 €) nach Vorbestellung bei der LMU- und der TU-Bibliothek gibt. Nicht unbedingt für Neueinsteiger geeignet.

Lars Heppert, *Coding for Fun*, Galileo Computing, 1. Auflage 2010, 330 S, 24,90 €. Der Einband verspricht „Garantiert kein Lehrbuch!“, stattdessen werden kleine Spaßapplikationen (3D-Uhr mit OpenGL und pyGame, das Spiel „Snake“, Verschlüsselungen, Webcrawler, ...) mit Python entwickelt und daran unaufdringlich Software- und Informatikkonzepte erklärt (Refactoring, Markov-Ketten, Suchbäume, ...). Nicht für blutige Anfänger!

1.3.2. Python für wissenschaftliche Anwendungen

Software Carpentry, http://software-carpentry.org/v4/index.html, bietet neben allgemeinen Kursen zu Python (s.o.) auch die spezielleren Themen „Matrix Programming“ und „*Matlab*“ an.

Valentin Haenel, Emmanuelle Gouillart, Gaël Varoquaux, *Python Scientific lecture notes Release 2012.3*, EuroScipy 2012, 309 S., http://scipy-lectures.github.com/_downloads/PythonScientific-simple.pdf aus http://scipy-lectures.github.com. Die Unterlagen geben auf 335 Seiten einen Überblick über die wichtigsten wissenschaftliche Module in Python (numpy, scipy, matplotlib, ...).

Matlab-Python Spickzettel, http://mathesaurus.sourceforge.net/matlab-numpy.html.

Hans Petter Langtangen, *A Primer on Scientific Programming with Python*, Springer Verlag, 798 S., 3. Auflage 2012, Ca. 60 €. Sehr gut strukturiert, viele Beispiele aus verschiedenen Wissenschaftsgebieten, wird ständig aktualisiert.

Achtung: In manchen Unterlagen wird noch das Python-Modul `math` verwendet, das für schnelle Array - Operationen durch `numpy` ersetzt werden sollte.

2. Woher nehmen?

2012 hat Python für wissenschaftliche Anwendungen Fahrt aufgenommen; man hat die Auswahl zwischen einigen freien und kommerziellen Distributionen. Tabelle 2.1 listet eine Auswahl auf; bis auf WinPython unterstützen die Distributionen „alle“ Betriebssysteme, d.h. Windows, Linux, Mac OS X. Im akademischen Umfeld dürfen die Vollversionen von Enthought und Continuum Analytics kostenfrei eingesetzt werden, beide Firmen sind auch sehr aktiv in der Python Open-Source Community.

Vor allem Einsteiger sollten *unbedingt* die ersten Schritte im geschützten Raum einer Distribution mit aufeinander abgestimmten Modulen zu machen. Ein Zitat von Ende 2012 von Fernando Perez, einem der Hauptentwickler von IPython, bringt es auf den Punkt: [http://ivory.idyll.org/blog/teaching-with-ipynb-2.html]:

> [...] packaging/distribution in python is an unmitigated, absolute disaster of cosmic proportions. [...]

Tab. 2.2 listet ein paar IDEs für Python auf, ich habe mich hier auf IDEs beschränkt, die frei verfügbar für die „großen“ Betriebssysteme sind und mir gefallen haben. Ich habe dabei unterschieden nach

Wissenschaftlichen IDEs, die der Funktionalität von Matlab^{TM} nachempfunden sind. Am Wichtigsten sind hier die Möglichkeiten, Array- und Matrix-Datentypen von Numpy in einem Variable-Explorer darzustellen und ein guter Zugriff auf die Dokumentation von numpy, scipy und matplotlib. Manche IDEs bilden die „Cells“ Funktionalität von Matlab^{TM} nach (der Code wird durch Kommentarzeilen, die mit ## beginnen, in Zellen unterteilt, die einzeln ausgeführt werden können).

Allgemeine IDEs, deren Stärken eher im Management komplexer Software-Projekte liegen (Projektverwaltung, Versionsmanagement, Code testing, ...)

Shells und Editoren, die nur Basisfunktionen wie Syntax-Highlighting, automatische Einrückung (wichtig für Python!), eine interaktive Shell etc. bieten.

Die Grenzen sind natürlich fließend. Ein umfassenderer Vergleich findet sich unter: http://en.wikipedia.org/wiki/List_of_integrated_development_environments_for_Python#Python.

2.1. Python 2 oder 3?

Die Python 2.x Serie wurde im Juli 2000 eingeführt, mit 2.7 wurde im Juli 2010 die letzte Version der 2.x Serie veröffentlicht, für die es aber noch über einen längeren Zeitraum Bugfixes geben wird. Aktueller Stand (Feb. 2017) ist die Version 2.7.12. In Python 2.7 wurden viele Features der Version 3.1 zurückportiert, Code der unter 2.7 entwickelt wurde und die neuen Features nutzt, lässt sich daher leicht auf die 3.x Serie portieren. Details siehe http://www.python.org/download/releases/2.7/.

Wissenschaftliche Distributionen	
WinPython	Freie kompakte Distribution nur für Windows, auch 64 Bit Version und auch für Python 3, portabel und mit eigenem Package Manager, mit Spyder als Standard-IDE. Winpython ist aus Python(x,y) entstanden.
Python(x,y)	Umfangreiche freie wissenschaftliche Distribution, die allerdings nur in größeren Abständen aktualisiert wird. Nur für Windows, nur 32 bit, nur Python 2.7.
Anaconda	Sehr gut ausgestattete Distribution für wissenschaftliche Anwendungen von Continuum Analytics. Der freien „Community Edition“ Anaconda CE fehlen im Vergleich zur kommerziellen Edition nur ein paar spezielle Module für „Big Scale Data“. Mayavi und Traits fehlen
Canopy	Speziell für den wissenschaftlichen Einsatz ausgestattete (kommerzielle) Distribution von Enthought mit interaktivem Python Desktop, die benutzerfreundliche Bedienung und Updates haben soll. Genau wie bei EPD (s.u.) fehlt Spyder und lässt sich anscheinend auch nur schwer nachinstallieren.
Allgemeine Distributionen	
Ubuntu	ist natürlich erst einmal eine Linux-Distribution, die aber „out-of-the-box“ alle benötigten Python-Tools mitbringt (oder leicht nachinstallieren lässt)
EPD	(Enthought Python Distribution): Kommerzielle „Schweizer Taschenmesser“-Distro, ebenfalls von Enthought. Die allgemeine freie Distro ist stark beschnitten.

Tab. 2.1.: Python Distributionen

Wissenschaftliche IDEs	
Spyder	(**S**cientific **PY**thon **DE**velopment **E**nvi**R**onment) gewinnt den Preis für „Matlab™-Ähnlichkeit“. Variable Explorer zum direkten Zugriff auf NumPy Arrays, Dictionaries, Lists etc., gemeinsame Version für Python 2.7 und 3.3 ab Version 2.3
iep	(Interactive Editor for Python), eine aufgeräumte Entwicklungsumgebung mit „Cells“ (nur Python 3.3)
Allgemeine IDEs	
pydev	Plugin für Eclipse - wer bereits mit Eclipse arbeitet, sollte das Plugin mal testen.
PTVS	(Python Tools for Visual Studio) ist ein OSS Plugin von Microsoft für Visual Studio (nur Windows), zumindest die Visual Studio Shell kann umsonst benutzt werden. Wer eh mit Visual Studio arbeitet, sollte das Plugin mal ausprobieren: „One of the least-known and most-kick-butt free products we [Microsoft] have is PTVS“ [http://www.hanselman.com/blog/OneOfMicrosoftsBestKeptSecretsPythonToolsForVisualStudioPTVS.aspx].
Interaktive Shells und Editoren	
IPython	Interactive Python - neben verschiedenen interaktiven Shells (Terminal, Qt, ...) kann man mit IPython auch im Web-Browser arbeiten und dort remote oder lokal webbasierte Notebooks bearbeiten und ausführen. Auf der Homepage http://ipython.org finden sich auch Tutorial-Videos und Beispiele.
sublime	ist ein sehr leistungsfähiger Editor (70 $ nach der zeitlich unbegrenzten Evaluationsphase) mit vielen Plugins
IDLE	Das vi unter den Python IDEs: Wird defaultmäßig mit jeder Distribution mitgeliefert, geht immer. Hat allerdings ein eher unübliches Bedienkonzept und verschiedene Usability-Probleme ("‘The Things I Hate About IDLE That I Wish Someone Would Fix"‘). Mit den Erweiterungen IdleX werden viele Schwächen behoben.

Tab. 2.2.: Integrierte Entwicklungsumgebungen (IDEs) und interaktive Shells für Python

Seit Anfang 2009 ist Python 3 verfügbar (Stand Feb. 2017: 3.6.0), Ziel war es viele kleine „Unzulänglichkeiten“ und „Ungereimtheiten“ von Python 2.x zu beseitigen, auch wenn das bedeutete, dass Python 3 Code nicht direkt kompatibel zu 2.x ist - in beiden Richtungen.

Die Scientific Community hat relativ langsam auf Python 3 umgestellt, inzwischen sind aber praktisch alle Pakete unter Python 3 verfügbar. Für neue Projekte sollte man daher immer auf Python 3 setzen.

Um den Umstieg zu erleichtern, gibt es Trostpflaster:

- `from __future__ import ...` rüstet einen Teil der Python 3 Funktionalität für Python 2 nach (s.u.)
- `import six` importiert die Kompatibilitätsbibliothek `six` (zwei x drei = sechs ...)
- `2to3` ist ein Skript, mit dem man Code auf Kompatibilität zur 3.x testen und wahlweise z.T. automatisch übersetzen kann.

2.1.1. Print-Funktion

Eine Änderung, über die man sofort stolpert, betrifft „`print`“: Unter Python 2 ist `print` ein Statement, unter Python 3 eine eingebaute Funktion. Für die String-Formatierung wurden zusätzliche Optionen geschaffen:

```
1 print 'hello, world!'   # Python 2
2 print('hello, world!')  # Python 3
3 # Ausgabe von "pi = 3.14 und e = 2.718e+00" mit Zeilenumbruch:
4 from numpy import e, pi
5 print 'pi = %1.2f und e = %1.3e\n' % (pi, e)              # Python 2
6 print('pi = %1.2f und e = %1.3e\n' % (pi, e))            # Python 3
7 print('pi = {0:1.2f} und e = {1:1.3e}\n'.format(pi, e))  # Python 3
```

Dieses Verhalten lässt sich unter Python 2 mit `from __future__ import print_function` erzielen.

2.1.2. Standardeingabe

Ebenfalls verändert wurde der Zugriff auf die Standardeingabe: Unter Python 2 liest `raw_input` Text von der Standardeingabe ein und gibt ihn in Rohform zurück. Die Funktion `input` liest ebenfalls Text von der Standardeingabe ein, interpretiert ihn aber sofort als Python Code und führt ihn aus. Unter Python 3 gibt dagegen die Funktion `input` Text in Rohform zurück, Interpretation wird durch `eval(input())` erzwungen. Das ist zwar logischer, verwirrt aber beim Umstieg.

2.1.3. Integerdivision

Eine „versteckte“ Falle ist die Änderung der Integerdivision: In Python 2.x ergibt `1/2` den Wert 0 und `1./2` den Wert 0.5 (ein fieser Stolperstein für Matlab™-Umsteiger ...). In Python 3.x ergibt auch `1/2` den Float 0.5, Integerdivision muss durch `1//2` spezifiziert werden (auch in 2.x verfügbar). Dieses Verhalten kann man auch der 2.x Reihe beibringen durch die Anweisung `from __future__ import division` .

2.1.4. Absolute und relative Importe

Python 3 - Verhalten kann erreicht werden mit `from __future__ import absolute_import` .

2.1.5. Strings? Unicode?? ASCII???

http://nedbatchelder.com/text/unipain.html
http://stackoverflow.com/questions/4545661/unicodedecodeerror-when-redirecting-to-file

```
b'my bytes'  # Python 3, available in Python >= 2.6
u'my string' # Python 2, available in Python >=3.3
```

Eine vollständige Liste der Änderungen findet sich unter http://docs.python.org/py3k/whatsnew/3.0.html, gute Beispiele dazu unter http://spartanideas.msu.edu/2014/06/01/the-key-differences-between-python-2-7-x-and-python-3-x-with-examples/.

Mittlerweile sind die meisten Module für wissenschaftliche Anwendungen nach Python 3 portiert worden, das war Anfang 2014 noch nicht so. Immer mehr Module werden jetzt zuerst für Python 3 und dann für Python 2 entwickelt. Ich würde daher inzwischen empfehlen, mit Python 3 zu arbeiten.

2.1.6. Faulheit siegt: Generatoren und Listen

Python 2: `range(...)` liefert eine Liste (alle Elemente werden erzeugt und im Speicher abgelegt), `xrange(...)` liefert ein Sequenzobjekt, das „lazily“ evaluiert wird, d.h. es generiert nur dann einen Wert wenn er gebraucht wird. Bei großen Werten wird so deutlich weniger Speicher gebraucht.

In Python 3 ist dieses Verhalten Default, hier verhalten sich `range` und `xrange` identisch. Will man Python 2 Verhalten von `range` in Python 3, muss man schreiben `list(range(...))`.

2.1.7. Aufräumen

Achtung: Beim Umstieg von Python 2 auf Python 3 beschwert sich der Python-Interpreter u.U. mit der kryptischen Fehlermeldung „Bad magic number“, da die kompilierten `*.pyc` Files mit der falschen Version kompiliert wurden. Man löscht rekursiv alle `*.pyc` Files mit
`del /S *.pyc` (Windows) bzw.
`find . -name '*.pyc' -delete` (*nix)

jeweils von der Console aus. Durch Voranstellen von `!` kann die Kommandos auch aus einer Python-Konsole absetzen.

Die Erstellung von `*.pyc` und `*.pyo` Files kann auch komplett unterdrückt werden, indem man in

`site-packages/usercustomize.py` setzt `PYTHONDONTWRITEBYTECODE=1`.

Den gleichen Effekt hat es den `-B` Switch beim Aufruf des Python Interpreters zu übergeben.

Der Wechsel von Python 3 auf Python 2 ist unproblematisch, da die Bytecodefiles in Python 3 in einem eigenen Ordner `__pycache__` abgelegt werden.

Speziell wenn man Pfade und Modulstrukturen geändert hat, hilft es bei seltsamen Fehlermeldungen machmal, *alle* Bytecodefiles zu löschen.

2.2. Von der Quelle

Zum Beispiel für Numpy:

```
pip install -e .
git pull && python setup.py build_ext -i
```

2.2.1. Selber Anbauen

Zum Hochladen einer eigenen Software auf PyPI benötigt man vor allem ein gut gepflegtes `setup.py` zusammen mit anderen empfohlen Files (`MANIFEST.in, README, LICENSE`).

Damit kann man relativ leicht ein Paket erstellen, das über PyPI und (mit geringem Zusatzaufwand) und Anaconda verteilt werden kann, siehe https://packaging.python.org/distributing/

First, do a test run on testpypi.python.org/pypi, specifying the repository (-r):

```
python setup.py register -r https://testpypi.python.org/pypi
python setup.py sdist
upload -r https://testpypi.python.org/pypi

# Now, do the "'real thing"':
python setup.py register
python setup.py clean
```

```
#  create source package as .tar.gz :
python setup.py sdist upload
#  create source package in zip format
python setup.py sdist  --format=zip upload
# create binary wheel package - repeat for py2 / py3
python setup.py bdist_wheel upload



#Change to a new directory and try to install:
pip install -i https://testpypi.python.org/pypi <package name>

python setup.py register
python setup.py clean
python setup.py sdist
python setup.py sdist upload

pip install <package name>
```

Die Variante `bdist_wheel` ist optional.

2.3. Noch mehr Varianten

„Python“ wird synonym als Begriff für die Programmiersprache und für die „traditionelle“, in C geschriebene Implementierung von Python (auch CPython genannt) verwendet. Neben CPython gibt es unter http://www.python.org u.a. auch die folgenden alternativen Implementierungen:

IronPython: Python in einer .NET Umgebung

Jython: Python auf einer Java Virtual Machine

PyPy: Eine schnelle, in Python geschriebene Implementierung von Python mit Just-In-Time Compiler. Nicht zu verwechseln mit PyPI, dem Python Package Inventory.

Diese Varianten hinken meist einige Versionen hinter CPython her, es lassen sich auch längst nicht alle Python-Module importieren. Sie sollten sie daher nur einsetzen, wenn Sie genau wissen warum.

3. Distributionen

3.1. WinPython

WinPython (https://winpython.github.io/ ist eine portable Open Source Python-Distribution für Windows, die optimal für den Einstieg und zum Ausprobieren geeignet ist: Einfach Herunterladen und in einem beliebigen Directory (möglichst ohne Blanks im Pfad) oder auf einem USB-Stick installieren. WinPython ändert zunächst keine Systemeinstellungen wie die Pfadvariable oder die Registry, man muss daher Skripte vom WinPython Command Prompt (s.u.) aus starten oder den Pfad zum Pythoninterpreter angeben. Über das Winpython Control Panel (s.u.) kann Winpython aber jederzeit „registriert“ werden; das heißt hier *nicht* sich irgendwo für Werbemails anzumelden, sondern Pfade etc. in der Windows *Registry* einzutragen! Das kann genauso schnell wieder rückgängig gemacht werden.

Eine Kurzeinführung in die Installation von WinPython zeigt http://www.youtube.com/watch?v=iCWo8JR9g4Y. Erste Schritte mit Python http://www.youtube.com/watch?v=BWjLnpMQhRs (sehenswert!)

Im Installationsdirectory von WinPython finden sich Launcher für die wichtigsten Executables:

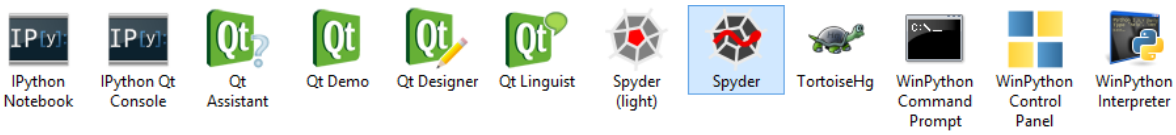


Abb. 3.1.: WinPython Launchers

WinPython Control Panel / WinPython Packet Manager: Kleines GUI zum Installieren und Deinstallieren von zusätzlichen Modulen. Normalerweise würde man das mit den Standard-Installationsskripten wie `easy_install`, `pip` oder `distutils` machen, das klappt aber nicht ohne Verankerung in der Registry. Der WinPython Packet Manager (WPPM) kann „reine” Python-Module im gezippten Format (`foo-1.0.tar.gz`) oder im „Wheel“-Paketformat oder solche mit Binärfiles (`mein_modul-1.0.win32-py2.7.exe`) installieren, man braucht daher keinen C/C++ Compiler. Eine gute Quelle hierfür ist Christoph Gohlke's Sammlung.

IPython Notebook: Startet IPython im lokalen Webbrowser, damit können interaktive Notebooks erstellt und bearbeitet werden.

IPython Qt Console: Die interaktive Konsole, die auch matplotlib Plots anzeigen kann.

Qt Designer: Werkzeug zum Entwurf von grafischen Oberflächen mit Qt

Spyder: Ruft die Entwicklungsumgebung Spyder (s.u.) auf. Die „light“ Version enthält nur Console und Variable Explorer.

WinPython Command Prompt: Ein „normales“ DOS Command Fenster, bei dem aber die Pfadvariable bereits die verschiedenen Python-Directories von WinPython enthält. Pythonskripte, z.B. zur Installation von Modulen (s.u.) sollte man also aus diesem Fenster heraus starten.

Über Advanced -> Register Distribution wird WinPython in der Registry und in der Pfadvariable „fest verankert“, d.h. es werden Einträge im Start- und Kontextmenü erzeugt und Python-Files mit den entsprechenden Programmen verknüpft. Genauso einfach können diese Eintragungen auch wieder entfernt werden.

WinPython Interpreter: Startet eine einfache Konsole mit dem Python Interpreter

3.1.1. Nach der Installation

Achtung: Da WinPython nur minimal in die Systemeinstellungen eingreift (vor allem wenn man es nicht „registriert“ hat), gibt es oft Probleme mit parallel installierten und in der Registry verankerten Python-Distribution wie pythonxy, Anaconda etc. Im File `winpython.ini` lassen sich dann unter

```
[environment]
## <?> Uncomment lines to override environment variables
#PATH =
#PYTHONPATH =
#PYTHONSTARTUP =
```

die entsprechenden Pfade für die Laufzeit - also nicht dauerhaft - ändern.

3.1.2. Matplotlib

3.1.3. pyreverse / GraphViz

Um -> pyreverse nutzen zu können, muss der Pfad zum Interpreter in `settings\winpython.ini` eingetragen werden:

```
PATH = C:\Program Files (x86)\Graphviz2.38\bin
```

3.1.4. pygmentize

Der Syntax-Highlighter **`Pygments`** wird bereits mit Winpython mitinstalliert und z.B. in Spyder verwendet. In einer Latex-Distribution kann er ebenfalls verwendet werden, wenn „pygmentize.cmd“ bzw. „pygmentize.exe“im Pfad gefunden wird:

```
@echo off
set PYTHONPATH=C:\Python27
%PYTHONPATH%\python.exe %PYTHONPATH%Scripts\pygmentize %*
```

Alternativ können beide Pfade zur Pathvariable hinzugefügt werden.

3.2. Anaconda

Die Distribution Anaconda (https://store.continuum.io/cshop/anaconda/) wird von Continuum Analytics für Python 2.7 und 3.4 entwickelt und überwiegend kostenfrei zur Verfügung gestellt.

Die Pakete **numba** und **numbaPro**, (siehe Kap. C) werden von Continuum Analytics entwickelt und können optional eingebunden werden. Kostenpflichtige Pakete wie numbaPro sind für Nutzer mit Academic Lizenz ebenfalls kostenlos.

Als Entwicklungsumgebung ist defaultmäßig Spyder installiert.

3.2.1. Installation

Siehe https://docs.continuum.io/anaconda/install

Windows

- Installation in einen noch nicht existierenden Ordner mit dem Namen „Anaconda3“ (möglichst)
- „Add Anaconda to my PATH Variable“ / Register Anaconda as my default Python 3.5“: Genau wie Winpython könnten Sie Anaconda „light“ installieren, d.h. ohne Spuren im System zu hinterlassen. Wenn Sie nicht eine weitere Python-Installation parallel nutzen wollen, sollten Sie beide Optionen aktiviert lassen.

3.2.2. Paketmanager Conda

Anaconda enthält den Paketmanager **conda** (http://conda.pydata.org/docs/intro.html), der Pakete mitsamt Abhängigkeiten installieren, updaten und deinstallieren kann. Neben den offiziellen Paketquellen von Continuum Analytics kann man weitere „Channels“ (URLs) mit spezielleren Paketen abonnieren.

`Conda` ist außerdem ein „Environment Manager“, mit dem man schnell zwischen Umgebungen (= eigene Directories) mit unterschiedlichen Python- und Paketversionen wechseln kann, also vergleichbar mit `virtualenv`.

http://conda.pydata.org/docs/_downloads/conda-pip-virtualenv-translator.html stellt in einer Tabelle die Features und grundlegenden Befehle von `conda`, `pip` und `virtualenv` gegenüber.

Der Anaconda Navigator ist ein Desktop GUI, der diese Aufgaben auch Clicki-Bunti ohne Kommandozeile erledigt. Momentan (Juni 2016) steht allerdings noch nicht die volle Funktionalität von conda zur Verfügung.

Rufen Sie conda aus der Shell bzw. einem DOS-Fenster auf; das Verzeichnis Anaconda/scripts sollte im Pfad eingetragen sein. Wichtige Befehle sind u.a.

```
conda help
conda info
conda list
conda search
conda install <package>
conda update <package>
```

`conda update -all python=3.5` aktualisiere alle Pakete auf Python 3.5 `conda clean -all` ... und räume auf

`anaconda search -t conda pyfda` suche in der Anaconda Cloud nach dem Paket pyfda

`anaconda show <USER/PACKAGE>` für mehr Details

conda install –channel https://conda.anaconda.org/Chipmuenk pyfda

3.2.3. Eigene Kanäle

Pakete, die nicht von conda oder Anaconda.org verwaltet werden, aber unter https://pypi.python.org/ zur Verfügung stehen, können mit **pip** problemlos nachinstalliert und upgedated werden, da `pip` bereits in Anaconda vorinstalliert ist. Beispiel für das Paket `pyfda` : `pip install pyfda`

Noch komfortabler ist es, Software aus „Channels“ von Usern zu installieren über

Liegt ein Paket bereits auf PyPI vor, ist es relativ einfach daraus ein Anaconda Paket zu machen (http://conda.pydata.org/docs/build_tutorials/pkgs.html)

Install conda-build:

conda install conda-build

It is recommended that you use the latest versions of conda and conda-build. To upgrade both packages run:

conda upgrade conda conda upgrade conda-build

It is easy to build a skeleton recipe for any Python package that is hosted on PyPI, the official third-party software repository for the Python programming language.

In this section you are going to use conda skeleton to generate a conda recipe, which informs conda-build about where the source files are located and how to build and install the package.

First, in your **user home directory**, run the conda skeleton command:

`conda skeleton pypi pyfda`

The two arguments to conda skeleton are the hosting location, in this case **`pypi`** and the name of the package (**`pyfda`**).

This creates a directory named `pyfda` and creates three skeleton files in that directory: `meta.yaml`, `build.sh`, and `bld.bat`. Use the ls command on OS X or Linux or the dir command on Windows to verify that these files have been created. The three files have been

populated with information from the PyPI metadata and in most cases will not need to be edited.

These three files are collectively referred to as the conda build recipe:

meta.yaml: Contains all the metadata in the recipe. Only the package name and package version sections are required; everything else is optional.

bld.bat: Windows commands to build the package.

build.sh: Linux and OS X commands to build the package.

Now that you have the conda build recipe ready, you can use the conda-build tool to create the package.

```
conda build pyfda
```

When conda-build is finished, it displays the exact path and filename of the conda package. See the Troubleshooting section if the conda-build command fails.

3.2.4. Navigator

3.2.5. Environment

Ähnlich wie **virtualenvs** lassen sich unter Anaconda separate Umgebungen einrichten, in denen man mal schnell ein Paket oder eine neue Pythonversion testen kann.

Wichtige Befehle sind u.a.

```
conda info -envs
```

```
source deactivate
```

Beim Einrichten eines neuen Environments kann die Version von bestimmten Paketen genau vorgegeben werden mit z.B. `numpy=1.10` oder

`conda create -n py35 python=3.5 anaconda` ... erzeugt ein neues Environment mit dem Namen „py35“ und Python 3.5, dabei ist „anaconda“

3.3. Ubuntu

... wird von manchen als die beste Python-Distribution bezeichnet, da alle wichtigen Python-Module bereits enthalten sind und der Rest sich leicht nachinstallieren lässt. Siehe auch https://wiki.ubuntu.com/Python/3

4. Entwicklungsumgebungen

4.1. RunRunRun

Da Python eine interpretierte Sprache ist, muss man nicht kompilieren, sondern kann Skripte sofort ausführen:

4.1.1. Einfach laufen lassen

Um ein Pythonskript unter Windows auszuführen, müssen der Pfad zu Python-Interpreter und Skript angegeben werden. Wenn der Pythoninterpreter in der Path-Variable eingetragen ist, genügt natürlich `python mein_skript.py`. Unter der Winpython Distribution gibt es den „WinPython Command Prompt“, eine DOS - Commandobox, bei die Python - Pfade bereits definiert sind. So muss man die Pfade nicht hart eintragen und kann z.B. mehrere Pythonversion parallel auf dem Rechner betreiben.

Bei dieser Variante kann man dem Python-Interpreter Optionen mitgeben wie z.B. `-verbose-debug` um z.B. mehr Infos bei seltsamen Fehlern zu bekommen.

Man kann ein Pythonskript auch mit der `interactive` Option ausführen; dann landet man nach Beendigung des Skripts in der REPL (s. nächster Abschnitt) und hat z.B. Zugriff auf Variablen und importierte Module.

```
python -i mein_skript.py
```

4.1.2. Aus dem Programm

Aus dem Python Interpreter heraus startet man ein Skript mit

```
execfile("mein_skript.py")
```

In IPython gibt es außerdem das Makro `%run`:

```
% run mein_skript.py
```

4.1.3. Run Optionen

`-c` **`command:`** Führe übergebenes Kommando direkt aus.

`-h` **`help:`** Zeige alle verfügbaren Optionen an

`-m` **`module:`** Suche in `sys.path` nach dem angegebenen Modul und führe es als `__main__` aus. Da das Argument der Modulname ist, darf die Endung `*.py` nicht angegeben werden! Wenn ein Paketname anstelle eines Modulnamens übergeben wird, wird `pkg.__main__` ausgeführt (falls es existiert). Beispiel: `python -m timeit -c 3/4`.

`-q` **`quiet:`** Unterdrücke Copyright- und Versionsmeldung

`-V` **`Versionnumber`**

`-v` **`verbose:`** Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. When given twice (-vv), print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit. See also PYTHONVERBOSE.

`-X` Verschiedene implementierungsspezifische Optionen zum Debugging: `-X faulthandler:` Enable faulthandler (CPython ab Version 3.3), bei :

4.1.4. REPL und IDLE

Auf den Begriff **REPL** (read–eval–print loop) stößt man in Python sehr oft. Gemeint ist, dass man sich nach dem Starten einer Python Shell (mit `python`, ggf. mit voransgestelltem Pfad) interaktiv den Interpreter bedienen kann:

Read liest User Inputs und parsed sie in eine Datenstruktur im Speicher.

Eval evaluiert diese interne Struktur.

Print druckt das Ergebnis des letzten Schritts aus und formatiert es gegebenenfalls.

Loop fängt von vorne an und wartet auf eine neue Eingabe des Nutzers.

Das kann im einfachsten Fall `>>> print(2+3)` sein (`>>>` stellt die Eingabeaufforderung des Interpreters dar), es sind aber auch zusammengesetzte Kommandos möglich:

```
1 >>> for i in range(10):
2 ...     print(i)
3 ...
```

druckt die Zahlen von 0 bis 9 untereinander. Dabei wird „`...`' vom Interpreter ausgegeben um anzuzeigen, dass man gerade eine zusammengesetzte Anweisung tippt. Ein <RETURN> ohne vorherige Eingabe führt die Anweisung dann aus. Bei Python darf man natürlich die Einrückung nicht vergessen.

IDLE (*Integrated DeveLopment Environment* oder *Integrated Development and Learning Environment*) ist das `vi` unter Python IDEs mit dem Komfort eines alten VW-Käfers / Trabbis: unbequem, aber es läuft und läuft und läuft ... Es ist praktisch in jeder Python Distribution enthalten und ist wenig mehr als ein gedoptes REPL:

- Python Shell mit Syntax Highlighting
- Integrierter Debugger
- Multi-Window Texteditor mit Autocompletion

4.1.5. IPython

IPython ist u.a. eine interaktive Python - Shell, die durch eine Vielzahl von Makros und Hilfefunktionen sehr leistungsfähig ist (http://ipython.org/ipython-doc/stable/interactive/tutorial.html).

4.1.6. Module und Projekte

4.2. Spyder als Entwicklungsumgebung (IDE)

Spyder ist m.M. nach momentan die beste Python-Entwicklungsumgebung für wissenschaftliche Anwendungen, es ist leicht zu installieren und bietet u.a.:

Editor mit Syntax-Highlighting, auch für Matlab^{TM}-Code. Code-Completion wie üblich mit <CTRL-SPACE>,

Code-Introspection (De-Linting) d.h. Anzeige von einfachen Fehlern wie fehlenden Klammern und Verwendung undefinierter Variablen

Kontextsensitive Hilfe aus Editor und Konsole. Tipp: Die Hilfe wird angetriggert, wenn man die öffnende Klammer einer Funktion tippt oder <CTRL-I> über dem Argument bzw. dem Objekt.

Object und Variable Explorer

Debugger (pdb) mit grafischem Interface

Verschiedene Konsolen (u.a. IPython)

Profiling (welcher Teil des Codes verbraucht die meiste Rechenzeit)

Spyder ist in den meisten Distributionen (Ausnahme: Enthought Distributionen) für wissenschaftlichen Einsatz enthalten und wird daher normalerweise automatisch mit installiert. Unter Hilfe -> Optional Dependencies kann man überprüfen, ob alle Hilfsmodule für Syntaxcheck, Anzeige der Online-Hilfe etc. korrekt installiert sind.

Ein kurzes Tutorial zu Spyder finden Sie unter http://www.southampton.ac.uk/~fangohr/blog/spyder-the-python-ide.html

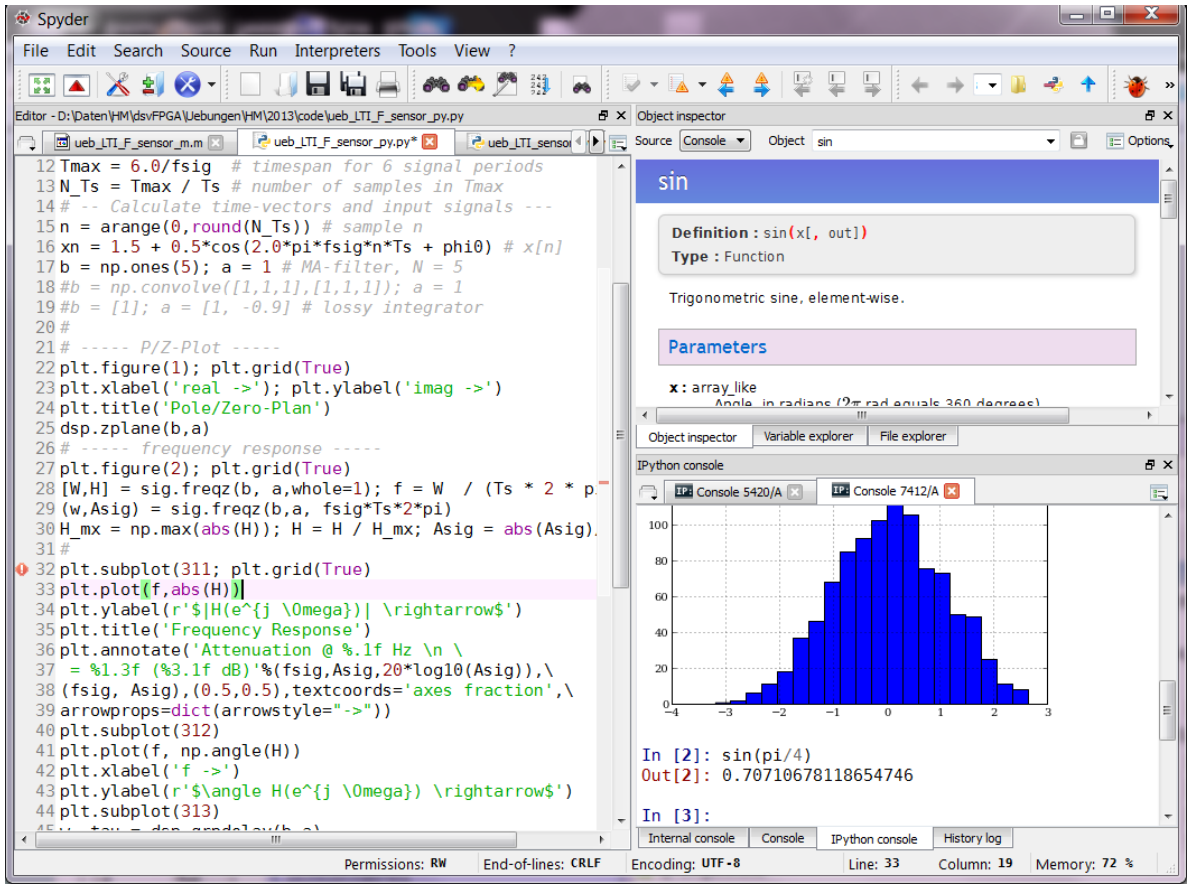


Abb. 4.1.: Screenshot der Entwicklungsumgebung Spyder 2.0 mit IPython Konsole und Inline-Grafik

4.2.1. Project Explorer

Spyder enthält eine rudimentäre Projektverwaltung, die sich über View -> Windows and Toolbars -> Project Explorer ein- und ausblenden lässt. Zunächst muss ein Directory als Workspace definiert werden, unter dem alle Projektdirectories liegen - danach wird automatisch gefragt, wenn man ein neues Projekt anlegen will.

Rechte Maustaste New Project oder Import Directory (z.B. aus geclonetem Git Repository)

4.2.2. Editor

Code completion: <CTRL>-<SPACE> (auch von Variablennamen) oder der Punkt nach nach dem Klassennamen
Context-sensitive Hilfe wird beim Tippen der geöffneten Klammer einer Funktion ausgelöst, <CTRL>-I zeigt die Dokumentation an.
Ab Version 2.3 kann über Tools > Preferences > IPython Console > Display > Interface das Verhalten der <TAB>-Taste (Auto-Completion Cycle) in IPython eingestellt werden.

4.2.3. Run Toolbar

Die verschiedenen Optionen, ein Skript aus dem Editor auszuführen, verhalten sich recht unterschiedlich; es lohnt sich daher sich damit ausführlicher zu beschäftigen.

Mit `Run -> Configure` oder <F6> kann man auswählen, wie das Skript beim Drücken von <F5> („Run“) ausgeführt werden soll:

Execute in current Python or IPython interpreter: Diese Einstellung führt das Skript immer im gleichen Interpreter aus, Variablen oder Fenster werden zwischendurch nicht gelöscht. Das ist das Gleiche wie `execfile('mein_script.py')` aus einer (I)Python Konsole heraus zu starten.

Ähnlich wie bei Matlab™, ist diese Einstellung praktisch zum Debuggen, u.U. merkt man aber nicht, dass man versehentlich eine Variablendefinition gelöscht hat. Über `Interpreters -> ...` kann man ggf. vorher eine neue Python oder IPython Konsole öffnen.

Execute in a new Python interpreter: Das ist das Gleiche wie `python mein_script.py` in einem Terminal zu starten. Wartet das Skript noch vom vorigen Programmstart z.B. in einem Matplotlibfenster, wird dies zuerst geschlossen, es wird also nicht jedes Mal eine weitere Python-Instanz gestartet.

Execute in an external system terminal: führt das Skript in einem neuen Terminal aus - keine Debugging-Möglichkeit, dafür auch keine Wechselwirkungen mit der interaktiven Konsole (siehe auch Kap. 4.2.6)

Nachfolgende F5 Befehle behalten das ausgewählte Verhalten bei.

4.2.4. Interaktive Codeausführung im Editor

<ENTER> führt die momentane Zeile aus <SHIFT>-<ESC> löscht die momentane Eingabezeile ohne sie auszuführen Pfeiltasten nach oben oder unten scrollen durch die History

4.2.5. Profiling

Mit <F10> bietet Spyder Zugriff auf den Standard-Profiler in Python, mit dem man herausfinden kann, welche Teile eines Programms am meisten Zeit kosten.

4.2.6. Skripte mit PyQt / PySide GUI in Spyder

Startet man Skripte mit PyQt oder PySide GUIs aus Spyder heraus in einer interaktiven Console, gibt es oft unerwartete Fehlermeldungen: In der interaktiven Konsole läuft bereits eine Qt Event Loop, außerdem ist die Konsole konfiguriert, um verschiedene wissenschaftliche Module zu importieren -tippe `scientific` am Spyder Konsolen Prompt für mehr Details.

Mit Run -> Configure -> Run in an external System terminal umgeht man diese Probleme, allerdings hat man auch keine Debugging-Möglichkeiten mehr und auch nicht den Komfort der interaktiven Konsole.

Um das Skript trotzdem innerhalb von Spyder zum Laufen zu bringen, muss man nur folgende Dinge beachten:

- Make sure Spyder is configured to open external consoles with PySide and not PyQt. This can be set from Tools>Preferences>Console>External modules>Qt-Python bindings library selection.
- With your script active in the editor, hit F6 to open the Run Settings dialog. Select the Ëxecute in a new dedicated Python interpreter"radio button instead of executing in the current interactive interpreter. Click OK. Now run the script by hitting F5. Debug the script by hitting Ctrl+F5.

4.2.7. Installation von Entwicklungsversionen

Hierzu lädt man den Source Code von der Projekt Website herunter (Source -> Browse -> Download zip), auspackt und `bootstrap.py` aus dem Hauptdirectory ausführt (siehe http://code.google.com/p/spyderlib/wiki/NoteForBetaTesters). Möchte man die Version installieren, kann man dies tun (ebenfalls aus dem Hauptdirectory) mit

```
python setup.py install
```

Bei WinPython muss man dazu den WinPython Command Prompt verwenden und in das entsprechende Verzeichnis navigieren. Sicherheitshalber sollte man vorher die alte Version von Spyder deinstallieren.

Wichtig: Die Module `rope` (Refactoring, Auto-Completion, holt Informationen von pydoc) und `pyflakes` (analysiert Python Programme „on-the fly“ and zeigt verschiedene Fehler an)

sollten installiert sein. Bei einer „regulären“ Spyder-Installation wird das automatisch erledigt, bei Installation vom Source Code (s.o.) muss man sich selbst darum kümmern. In WinPython erledigt man das am Einfachsten, indem man die Module herunterlädt und sie über das WinPython Control Panel installiert.

4.3. Jupyter / IPython

4.3.1. Root Directory

Aus Sicherheitsgründen kann man aus der Jupyter Oberfläche nur in Subdirectories wechseln, nicht aber nach oben in der Hierarchie. Daher ist es wichtig, das Default Root Directory für den Start von Jupyter sinnvoll festzulegen. Hierzu startet man aus einem Terminal

```
jupyter notebook --generate-config
```

Das erzeugt den File `.jupiter/jupyter_notebook_config.py` im Home Directory. In diesem File kann man die Zeile mit `# c.NotebookApp.notebook_dir = ''` aktiviern und editieren, z.B. als

```
c.NotebookApp.notebook_dir = 'D://Daten'
```

Das nächste Mal startet IPython aus diesem Directory.

Man kann das Notebook Directory auch für eine individuelle Kernel Session angeben mit

```
jupyter notebook --notebook-dir="D:\Daten\Design"
```

Man kann auch ein Command File (Windows) erzeugen, das das aktuelle Directory beim Start von Jupyter übergibt:

```
jupyter notebook --notebook-dir="%CD%"
```

4.4. Verwaltung

4.4.1. Was ist wo?

Durch Eingabe von `modules` am Python Prompt bekommt man eine Liste der installierten Module. Das dauert u.U. mehrere Minuten und ist ziemlich unübersichtlich, schneller geht's wenn man direkt im Directory `<install>/python-x.y.z/Lib/site-packages` nachschaut und sich z.B. mit `help('matplotlib')` weitere Informationen ausgeben lässt.

Um herauszufinden, welche Version eines Moduls an welchem Pfad installiert ist, hilft:

```
>>import numpy
>>print(numpy) # oder
>>print numpy.__path__
```

4.4.2. Installation von Modulen

Die meisten Distributionen sind von Haus aus recht gut ausgestattet und beinhalten einen Paketmanager (`conda` bei Anaconda, das Control Panel bei Winpython etc.), um weitere Module oder Updates zu installieren.

Von vielen komplexeren Modulen, die kompilierte C-Binaries enthalten, gibt es spezielle Binaries für die jeweiligen Betriebssysteme, die auf die „übliche“ Methode installiert werden (.exe oder .msi für Windows, RPM Pakete für Linux etc.). Quellen hierfür sind oft die Homepage des Projekts selbst oder die „Unofficial Windows Binaries for Python Extension Packages“ von Christoph Gohlke (`http://www.lfd.uci.edu/~gohlke/pythonlibs/`). Unter Linux / OS X wird man häufiger selbst Hand anlegen müssen.

Möchte man ein Python-only Modul von den Source-Files installieren, kann man das meistens mit dem `setuptools` tun:

setuptools

Nach dem Auspacken des zu installierenden Archivs (.zip oder .tar.gz) findet man im einfachsten Fall ein Setup Skript, das man ausführt mit

`setup.py install` bzw. `python setup.py install`

Für kompliziertere Fälle gibt es viele weitere Optionen, beschrieben in `http://docs.python.org/2/install/`.

Setuptools setzt auf dem Pythonmodul `distutils` auf.

4.4.3. pip

pip (pip installs Python) ist der empfohlene Python Installer:

```
pip install <package>
```

lädt das Package herunter (defaultmäßig vom PyPI, siehe unten) und installiert es. Weitere Optionen und Features unter `http://www.pip-installer.org`.

Siehe auch für einen kleinen Einblick in den Python Packaging Wahnsinn: `http://stackoverflow.com/questions/3220404/why-use-pip-over-easy-install`.

Das PyPI (Python Package Inventory) ist anscheinend gerade (Juni 2016) im Umbruch, Pakete sollte kann suchen unter `https://pypi.io/` oder `https://pypi.python.org/pypi`.

4.5. Flowchart Generierung / Reverse Engineering

Bei der Übernahme von fremdem Code (oder zur Dokumentation der eigenen Spaghettis) ist es nützlich, die Abhängigkeiten der einzelnen Module graphisch darzustellen. Das ist in Ansätzen möglich mit den folgenden Tools:

Visustin Automatic Flowchart Generator (`http://www.aivosto.com/visustin.html`), kommerzielle Lizenz 249$, academic license 149 $, Demomode verfügbar.

PyNSource Generiert UML-Diagramme aus Python-Code (`http://www.andypatterns.com/index.php`), per Kommandozeile oder GUI, Open Source. Die UML-Diagramme können leicht mit der Maus manipuliert werden, Codeteile können gelöscht werden. Diagramme können auch als ASCII-Art generiert und dann z.B. zu Dokumentationszwecken in eigenen Code kopiert werden.

PyUML ist ein Plugin für Eclipse (`http://sourceforge.net/projects/eclipse-pyuml/`) mit „Roundtrip-Funktionalität“ (Python -> UML -> Python).

pyReverse Generiert UML-Diagramme aus Python-Code, integriert in PyLint (`http://www.pylint.org/`). Skriptbasiert, Open Source. Dokumentation ist extrem spärlich, es muss außerdem `graphviz/dot` installiert sein (`http://www.graphviz.org/Download.php`), ein Tool zur Visualisierung von Graphen. Siehe nächster Abschnitt:

4.5.1. pyReverse

Der Charm von pyReverse liegt darin, dass es bei den meisten Python-Installationen bereits als Teil von pylint mit installiert ist. Allerdings muss das Graphikpaket Graphviz installiert und im Pfad eingetragen werden. Unter Winpython ist das einfach erledigt, indem man die Setting der Pfadvariablen anpasst: Unter `settings\winpython.ini` muss nur z.B. `PATH = C:\Program Files (x86)\Graphviz2.38\bin` ergänzt werden.

Wenn man dann das WinPython Command Prompt Fenster öffnet und `dot -Txxx` eingibt, sollte man die Fehlermeldung **„**`Format: "xxx"not recognized. Use one of ...`**“** bekommen.

Der einfachste Aufruf `pyreverse -o png -p myProj myProjDir`

erzeugt zwei Dateien **„**`classes_myProj.png`**“**“ und **„**`packages_myProj.png`**“**, die die Hierarchien im Verzeichnis `myProjDir` wiederspiegeln.

Weitere nützliche Optionen sind:

-a<n> oder -A : zeige `n` / zeige alle Generationen der „Ahnen“klassen

-c : Erzeuge ein Klassendiagramm mit allen Klassen ausgehend von <class>

-s<n> oder -S : zeige `n` / zeige alle Ebenen von assoziierten Klassen

-k : zeige nur die Klassennamen, nicht Attribute und Methoden

-myes oder -my : zeige den vollen Pfad des Modulnamen in der Klassendarstellung.

–ignore=file1,file2,... : ignoriere die aufgelisteten Files

Das Default-Outputformat ist `*.dot`, das mit dem Editor GVEdit (in Graphviz enthalten) editiert und in diverse Bitmap- und Vektorformate umgewandelt werden kann.

5. Eigenheiten von Python, Unterschiede zu Matlab™

5.1. Programmiersprache

Ein großer Vorteil von Python (und auch C++) ist die Unterstützung verschiedener Programmieransätze. Programme / Bibliotheken können überwiegend prozedural, objektorientiert oder funktional aufgebaut werden. In großen Programmen werden die Ansätze oft gemischt; so ist das GUI üblicherweise objektorientiert und die eigentliche Programmlogik oft prozedural oder funktional aufgebaut.

Bei **Funktionaler Programmierung** wird ein Problem in ein Set von Funktionen zerlegt. Reine Funktionen verarbeiten Eingangswerte und produzieren Ausgangswerte, sie haben keine Zustandsvariablen oder „Gedächtnis“ (siehe „Functional Programming HOWTO“ aus der Python Hilfe / Dokumentation).

Objektorientierte Programmierung mit Python ist anschaulich erklärt unter http://www.jeffknupp.com/blog/2014/06/18/improve-your-python-python-classes-and-object-oriented-programming/.

Die Eigenschaften und Vorteile von Python im Allgemeinen werden in Mike Levins Blogbeitrag“Python Programming Language Advantages“ (http://mikelev.in/2011/01/python-programming-language-advantages/) sehr anschaulich beschrieben.

Typische Anfängerfehler werden beschrieben in Constantine Lignos „Anti-Patterns in Python Programming“ (http://lignos.org/py_antipatterns/), Tipps für schnelleren Python-Code in Mark Litwintschiks „Faster Python“ (http://tech.marksblogg.com/faster-python.html).

5.2. Basics

Die Kombination aus Python, numpy, scipy und matplotlib wurde an vielen Stellen so gestaltet, dass der Umstieg für einen Matlab™ Nutzer nicht zu schwierig ist. Zu meiner eigenen Bequemlichkeit und zum leichteren Umstieg habe ich zusätzlich die häufigsten Befehle direkt in den Namespace importiert (siehe S. 40) Bei vielen Listings im „DSV auf FPGA“-Skript habe ich Python und Matlab™-Code Seite an Seite gesetzt so wie in Listing 5.1 und 5.2.

```
f1 = 50; Ts = 5e-3;
n = [0:49]; % sample n
t = 0:0.1:49; % start/step/stop
xn = 1.5 + 0.5*cos(2.0*pi*f1*n*Ts);
b = [0.1, 0]; a = [1, -0.9];
```

```
1 f1 = 50; Ts = 5e-3
2 n = arange(0, 50) # sample n
3 t = arange(0, 49., 0.1) # feiner aufgelöst
4 xn = 1.5 + 0.5*cos(2.0*pi*f1*n*Ts) # x[n]
5 b = [0.1, 0]; a = [1, -0.9] # Koeffizienten
```

```
%
[h, k] = impz(b, a, 30);
figure(1);
subplot(211);
stem(k, h, 'r-');
ylabel('h[k] \rightarrow'); grid on;
title('Impulse Response h[n]');
subplot(212);
stem(k, 20*log10(abs(h)), 'r-');
xlabel('k \rightarrow'); grid on;
ylabel('20 log h[k] \rightarrow');
% ------- Filtered signal ---
figure(2);
yn = filter(b,a,xn);
yt = interp1(n, yn, t, 'cubic');
hold on; % don't overwrite plots
plot(t, yt,'color',[0.8,0,0],'LineWidth',3);
stem([0:length(yn)-1],yn,'b');
xlabel('n \rightarrow'); grid on;
ylabel('y[n] \rightarrow');
title('Filtered Signal');
%
```

Lst. 5.1: Matlab-Codebeispiel

```
#   H(z) = (0.1 z + 0) / (z - 0.9)
[h, k] = dsp.impz(b, a, N = 30) # -> h[k]
figure(1)
subplot(211)
stem(k, h, 'r') # x[n], red stems
ylabel(r'$h[k] \rightarrow$'); grid(True)
title(r'Impulsantwort $h[n]$')
subplot(212)
stem(k, 20*log10(abs(h)), 'r')
xlabel(r'$k \rightarrow$'); grid(True)
ylabel(r'$20 \log\, h[k] \rightarrow$')
# ------- Filtered signal ------------
figure(2);
yn = sig.lfilter(b,a,xn) #filter xn with h
f = intp.interp1d(n, yn, kind = 'cubic')
yt = f(t) # y(t), interpolated
plot(t, yt, color='#cc0000', linewidth=3)
stem(n, yn, 'b') # y[n]
xlabel(r'$n \rightarrow$'); grid(True)
ylabel(r'$y[n] \rightarrow$')
title('Filtered Signal')
plt.show()       # draw and show the plots
```

Lst. 5.2: ... umgesetzt in Python

Ein paar Unterschiede zwischen den beiden Versionen sind:

Zeile 1: Matlab-Zeilen, die nicht mit einem Semikolon abgeschlossen werden, geben das Ergebnis an der Konsole aus. Bei Python ist das Semikolon optional; um Zwischenergebnisse auszudrucken, wird eine `print` - Anweisung benötigt.

Zeile 2: In Python kann man Arrays nicht ganz so kompakt deklarieren wie in Matlab. Das letzte Element (hier: 49) ist in Matlab eingeschlossen, in Python per Default *nicht*. Und Achtung: n[1] indiziert in Matlab das *erste* Element, in Python das *zweite* - Python nutzt (das für meinen Geschmack praktischere) Zero-based Indexing.

Zeile 6: Das Kommentarzeichen ist ein `#` im Gegensatz zu Matlabs`%`.

Zeile 7: Nicht alle Matlab-Funktionen (wie hier die `impz` - Funktion) sind direkt in Python verfügbar; meist lässt sich aber schnell ein Ersatz finden oder selbst schreiben. Hier wurde die `impz` - Funktion in einem Hilfsforum gefunden, angepasst und in eigene Bibliothek `dsp` gesteckt

Zeile 11: Python unterstützt einen deutlich größeren Teil des Latex-Sprachumfangs als Matlab; wie in Latex selbst werden Inline-Formeln zwischen `$ ... $` geschrieben. Der String muss mit vorangestelltem `r'...'` als „raw“ deklariert werden, damit nicht Sequenzen wie `\tau` als Tabulator + „au“ interpretiert werden.

Zeile 19: Manche Befehle heißen anders, haben aber die gleiche Funktionalität.

Zeile 20: Und andere Befehle sind komplett anders implementiert, hier muss man besonders aufpassen: Der Befehl `resample` wird z.B. in Python mit einer DFT implementiert und in Matlab mit einer Polyphasenfilterung.

Zeile 27: In Python werden mit dem `show()` - Befehldie Plots ausgegeben und können dann in einer Event-Loop interaktiv bearbeitet werden. Diesen Befehl vergisst man gerne am Anfang ...

Eine ausführliche Tabelle mit Python <-> Matlab^{TM} Äquivalenten findet man unter:
http://www.scipy.org/NumPy_for_Matlab_Users,
beim Umstieg hilft: http://wiki.python.org/moin/MovingToPythonFromOtherLanguages.

5.3. Konvertierung Matlab^{TM} -> Python

Einen vollständigen, automatischen Konvertierer Matlab^{TM} -> Python gibt es leider nicht, aber verschiedene Hilfsskripte:

smop (Small Matlab tO Python compiler, https://github.com/victorlei/smop) erscheint mir momentan (2014) das „aktivste“ Projekt zu sein. `smop` erhält die Struktur des Matlab^{TM} Codes und wandelt ihn in Python Code um ohne Vollständigkeitsanspruch. Grafikbefehle werden z.B. noch nicht unterstützt, auch die Umstellung 0-based -> 1-based Indexing muss man selbst vornehmen. Die konvertierten Code-Beispiele sehen aber überzeugend aus.

OMPC (One Matlab Per Child, https://bitbucket.org/juricap/ompc/) wandelt Matlab^{TM} Code in Python-kompatiblen Code um, der dann allerdings von der OMPClib abhängt. Die OMPClib ist eine weitere Numerik - Library, die u.a. 1-based Indexing und Copy on Assignment der Matlab^{TM} Array Arithmetik nachbildet.

5.4. Unterschiede zwischen Python und Matlab^{TM}

5.4.1. Text

Einrückungen: Programmteile und zusammengesetzte Anweisungen werden nicht über Klammern o.ä. gruppiert, sondern durch Einrückungen. Dementsprechend gibt es auch kein `next` oder `end` Statement in `for` und `while` Schleifen (siehe z.B. Listing 5.4).

Kommentare: Kommentarzeichen in Matlab^{TM} `%` werden durch `#` in Python ersetzt. Längere Kommentare werden in Python durch `"""` oder `'''` eingerahmt (sog. Docstrings). Bei Funktionen und Klassen gehört es „zum guten Ton“ direkt nach der Definition einen Docstring einzufügen, der dann auch in der Kontexthilfe angezeigt wird.

Textencoding: Damit Umlaute und andere Sonderzeichen verwendet werden dürfen, muss die Art des Text-Encodings als sog. **encoding cookie** in der ersten Zeile angegeben werden, z.B.:

`# -*- coding: iso-8859-15 -*-` oder `# -*- coding: utf-8 -*-`

Print: Zeilen, die in Matlab^{TM} nicht mit einem Semikolon abgeschlossen werden, veranlassen eine Ausgabe ins Kommandofenster. In Python muss man hierfür explizit ein `print()` Kommando verwenden, ein Semikolon am Ende der Zeile ist optional. In beiden Sprachen können mehrere Statements getrennt durch Semikolons in eine Zeile geschrieben werden (auch wenn das nicht die Übersichtlichkeit verbessert). Achtung: Das `print`-Kommando in Matlab^{TM} druckt die aktuelle Figure aus oder speichert sie in einen File - zur expliziten Ausgabe von Text verwendet man hier das Kommando `disp()`.

Überlange Zeilen: Matlab^{TM} verwendet „ ... “ um überlange Zeilen umzubrechen, in Python kann man Zeilen in den meisten Fällen einfach so umbrechen. Ausnahmen sind Strings, bei denen der Interpreter nicht eindeutig entscheiden kann was gemeint ist. In diesen Fällen nimmt man einen „ \“ für die Fortsetzung der Zeile. Der Backslash darf nicht Teil eines Kommentars sein und er muss das letzte Zeichen der (physikalischen) Zeile sein.

5.4.2. Namen, Module und Namespaces

File- und Variablennamen: File- und Variablennamen in Matlab^{TM} dürfen nur aus alphanumerischen Zeichen und dem Unterstrich bestehen, das erste Zeichen muss ein Buchstabe sein. Python ist etwas entspannter, aber auch hier schadet es nicht diese Einschränkungen zu beachten. Wichtige Ausnahme: In Python werden Unterstriche am Anfang benutzt, um „private“ File- und Variablennamen zu kennzeichnen (s. nächster Punkt). Definiert das File eine Funktion, müssen in Matlab^{TM} Funktions- und Filename identisch sein. In Python gibt es diese Einschränkung nicht, hier kann ein File auch mehrere Funktionen enthalten, die dann per `import myfunction1 from myfile` eingebunden werden.

Private Variablen: Ein einfacher Unterstrich am Anfang wird benutzt, um „private“, interne Klassenvariablen, Methoden etc. zu kennzeichnen. Das ist aber im wesentlichen eine Konvention. Namen, die mit zwei Unterstrichen beginnen (und maximal einem Unterstrich aufhören), werden per „name mangling“ geschützt, so dass sie nicht unabsichtlich überschrieben werden.

Namespaces oder -scopes: In Matlab^{TM} stehen sofort die Kommandos aller installierten Toolboxen zur Verfügung, in Python werden die gewünschten Module mit eigenen *Namespaces* importiert, so können Variablen, Funktionen und Klassen aus unterschiedlichen Modulen eindeutig angesprochen werden. Gleichnamige Funktionen werden nicht unabsichtlich überschrieben / überladen und es bleibt transparent, woher die einzelnen Funktionen stammen.

Innerhalb eines Moduls löst Python Namespaces nach dem „LEGB“-Prinzip auf:

Local: Namen, die lokal innerhalb einer Funktion (def oder lambda) zugewiesen wurden (nicht global)

Enclosing Function: Namen, die in einer übergeordnete Funktion definiert wurden.

Global: Namen, die auf dem Top-Level des Modulfiles deklariert wurden oder in einer def als des Modulfiles als global deklarierte Variablen

Built-In: Namen innerhalb des Python-Sprachstandards sind von allen Modulen erreichbar.

Python Module können auf unterschiedliche Arten importiert werden:

```
import numpy                    # importiere NumPy Modul mit eigenem Namespace,
x = numpy.pi                    # Namespaces werden nicht vermischt

import numpy as np              # importiere Modul bzw.
import numpy.random as rnd      # Untermodul mit abgekürztem Namespace
x = np.pi * rnd.normal()        # EMPFOHLENE VARIANTE !!

from numpy import *             # Import aller Funktionen eines Moduls in den
                                # gemeinsamen Namespace (nur für interaktives
                                # Arbeiten empfohlen: "'besudelt"' den Namespace)

from numpy import pi,log10      # Import einzelner, oft benutzter Funktionen in den
x = pi * log10(1000)            # in den gemeinsamen Namespace - Kompromiss.
```

Um die Codeschnipsel in den Übungen kurz zu halten, wird immer der Header Listing 5.3 für die Pythonprogramme verwendet (und nicht mitabgedruckt):

```
#!/usr/bin/env python
# -*- coding: iso-8859-15 -*-
#============================================================================
# _common_imports_v3_py.py
#
# Einfaches Code-Beispiel zum Kapitel "xxx", Übungsaufgabe yyy
#
#
#
#
#
#
# (c) 2014-Feb-04 Christian Münker - Files zur Vorlesung "DSV auf FPGAs"
#============================================================================
from __future__ import division, print_function, unicode_literals # v3line15

import numpy as np
import numpy.random as rnd
from numpy import (pi, log10, exp, sqrt, sin, cos, tan, angle, arange,
                   linspace, zeros, ones)
from numpy.fft import fft, ifft, fftshift, ifftshift, fftfreq
import scipy.signal as sig
import scipy.interpolate as intp

import matplotlib.pyplot as plt
from matplotlib.pyplot import (figure, plot, stem, grid, xlabel, ylabel,
    subplot, title, clf, xlim, ylim)

import my_dsp_lib_v3 as dsp
#----------------------------------------------------------------------------
# ... Ende der Import-Anweisungen
```

Lst. 5.3: Import-Kommandos für schnelle Matlab^{TM} -> Python Umsetzungen (und generell bequemes Arbeiten mit Python)

5.4.3. HILFE!

Die einfachste Art, Hilfe zu bekommen ist meist die kontextsensitive Hilfe aus IDEs wie z.B. Spyder. Im `doc` Verzeichnis der Pythoninstallation finden sich unter Windows kompilierte `.chm`

Files, im Internet gibt es zu den jeweiligen Modulen ausführliche Helppages. Man kann man aber auch im interaktiven Modus an der Konsole bzw. in der Shell einfach Hilfe anfordern. Manche der hier aufgelisteten Funktionen funktionieren nur mit Numpy Arrays (ndarray).

Wo finde ich ... `numpy.lookfor('Was', module = 'DiesModul')` gibt alle Funktionen / Klassen aus im angegebenen Modul oder Modulen, die 'Was' enthalten. Defaultmodul ist `numpy`.

Wie funktionierst Du? `help(func)` gibt den Docstring der Funktion `func` aus. Genauso erhält man die Hilfe zu Attributen einer Variablen, z.B. `help(a.max)`. `help()` ohne Argument startet das Hilfesystem von Python, das man mit 'q' beendet.

Was bist Du? Aufgrund des Dynamic Typing ist nicht immer klar, von welchem Typ eine Variable ist; hier hilft `type(a)`. Bei Numpy Variablen liefert `a.dtype` Informationen über den numerischen Typ (int16, complex, float, ...).

Wie groß bist Du (numpy-Variablen)? Der Befehl `size(a)` oder `a.size` gibt die Gesamtzahl der Elemente eines Numpy-Arrays an, `np.shape(a)` oder `a.shape` gibt ein Tuple zurück mit der Form der Variable und `len(a)` die Länge der *ersten* Dimension. Mit `ndim(a)` oder `a.ndim` erfährt man die Anzahl der Dimensionen.

Was kannst Du? Der Befehl `dir(a)` gibt alle Attribute des Objekts / der Klasse / der Variable aus. `dir()` ohne Argument druckt eine Liste aller Variablen, Funktionen etc. aus.

Eigene Funktionen / Klassen: Damit auch eigene Funktionen / Klassen eine schöne kontextsensitive Hilfe in Rich Text bekommen, benötigt die Dokumentation einen „Docstring" im *reStructuredText* (reST) - Format (`http://docutils.sourceforge.net/rst.html`) verfasst werden, einem „ easy-to-read, what-you-see-is-what-you-get plaintext markup syntax and parser system". Die Dokumentation von großen Python-Projekten wird mit Hilfe von Sphinx (`http://sphinx-doc.org/`) aus den reST-Sources automatisch generiert, die Sphinx-Doku beinhaltet auch eine sehr gute Erklärung der reST-Syntax. Siehe auch `http://stackoverflow.com/questions/3898572/what-is-the-standard-python-docstring-format`. Literatur: Jan-Ulrich Hasecke, „Software-Dokumentation mit Sphinx" (`http://literatur.hasecke.com/sachbuecher/software-dokumentation-mit-sphinx`).

Docstring: Der Text im `reStructuredText` - Format, der angezeigt werden soll, steht in einem Kommentar (eingerahmt von drei Anführungszeichen `"`) mit beliebig vielen Zeilen direkt unter dem Funktionsheader

Einrückungen: Ähnlich wie bei Python, strukturieren Einrückungen den Text.

Formatierungen: Ähnlich wie bei anderen Markup-Formatierungen kann man hier z.B. mit `* ... *` und `** ... **` Text hervorheben und mit `--` Unterstreichungen (müssen mindestens so lang sein wie der Text) Überschriften generieren.

Keywords: Bestimmte Keywords werden von manchen Renderern erkannt und dann farblich hervorgehoben.

5.4.4. Variablen und Kopien

Weist man in Matlab^{TM} eine Variable mit `b = a` zu, wird im Speicher eine neue, unabhängige Kopie des Variableninhalts von `a` erzeugt („Copy on Assignment“, „Deep Copy“). Das ist bei Python nicht unbedingt so und kann zu tückischen Fehlern führen. Der Grund hierfür liegt darin, wie Python Variablen anlegt:

Mutable vs. Immutable Variables: In Python sind Variablen Referenzen („Labels“) auf Objekte im Speicher (ähnlich wie in Java?). Deren eindeutige ID lässt sich über die Funktion `id(a)` ausgeben.

Bei manchen zusammengesetzten Variablentypen - `list`, `array`, `dict`, `set` und `ndarray` (NumPy) - kann das *Objekt* bzw. Elemente davon „in-place“ modifiziert werden, z.B. mit `myList[2] = 3`. Diese Typen nennt man „mutable“ (= veränderlich). **Achtung:** Wenn mehrere Variablen auf das Objekt verweisen, sehen diese Variablen ebenfalls die Änderung! (-> „shallow copy“, s.u.).

Objekte der Variablentypen `int, float, string, tuple` können nicht modifiziert werden („immutable“). Die Anweisung `myString[2] = "'a"'` führt daher zu einer Fehlermeldung. Weist man solchen Variablen einen komplett neuen Wert zu, `myString = „alt“; myString = „neu“`, wird ein neues Objekt (`"'neu"'`) im Speicher angelegt, auf das die Variable `myString` referenziert. Auch das Anhängen von z.B. Strings mit `+=` erzeugt ein neues Objekt und ist daher langsam, deutlich schneller sind optimierte Methoden wie `.join()`. „Verwaiste“ Objekte (auf die nichts mehr referenziert) werden bei der nächsten Garbage Collection gelöscht.

Es ist auf den ersten Blick nicht immer klar ersichtlich, was beim Kopieren von Mutable Variables oder Teilen davon („Slices“) passiert, da es drei verschiedene Varianten gibt (siehe auch http://wiki.scipy.org/Tentative_NumPy_Tutorial -> Views and Copies):

Keine Kopie: Mutable Variables werden als Referenzen übergeben, daher erzeugt `b = a` lediglich zwei Referenzen (Labels) auf das gleiche Objekt, d.h. `id(b) == id(a)` oder kürzer `b is a` ist `True`.

Shallow Copy, View oder Pass-By-Reference: In Python haben (bei mutable variables, s.o.) die Anweisungen `a = [1,2,3]; b = a; b[0] = 0` zur Folge, dass `a == b == [0,2,3]` ist, da das Objekt im Speicher verändert wird, auf das sowohl `a` als auch `b` referenzieren („Shallow Copy“ / „Pass-By-Reference“).

Deep Copy oder Copy on Assignment: Soll eine unabhängige Kopie des Objekts erzeugt werden, muss man eine „deep copy“ erzwingen. Wenn man den gewünschten Variablentyp noch einmal explizit angibt, also z.B. `b = list(a)` oder `b = np.array(a)`, wird anscheinend ein neues Objekt erzeugt. Mutable Variables des NumPy Modules verhalten sich ansonsten etwas anders als die entsprechenden Standard-Variablentypen:

Standardtypen: Bei Standard-Variablentypen `list`, `array`, `dict`, `set` erzielt man eine Deep Copy, indem man eine Slice der gesamten Variable zuweist: `a = range(5); b = a[:]; b[2] = -7` ändert nur `b`, nicht `a`.

Achtung: Das funktioniert nur einen Level tief, bei z.B. Listen von Listen muss man `deepcopy` aus dem Standardmodul `copy` verwenden.

Numpy etc.: Bei Numpy legt `a = arange(5); b = a[:]; b[2] = -7` einen *View* auf die gleichen Daten an, hier wird auch `a`geändert!! Bei `ndarrays` aus dem Numpy-Modul erreicht man eine deep copy mit `b = a.copy()`. Das funktioniert dann auch mit mehrdimensionalen `ndarrays`.

Mehrfache Zuweisungen: `a = b = c = d = 7 + 3` weist allen Variablen den Wert 10 zu. `a, b = b, a + 1` berechnet zunächst alle Argumente auf der rechten Seite, und weist sie dann a und b zu (hier: zuerst `a = b`, danach `b = a + 1`). Die Anzahl der Argumente auf beiden Seiten muss gleich sein (Listing 5.4) und kann auch als Tupel oder Liste erfolgen (`(a, b) = ...` oder `[a, b] = ...`)

```
# Fibonacci Serie - die Summe von 2 Elementen bestimmt das nächste:
a, b = 0, 1
while b < 10:
     print a, b
     a, b = b, a+b # heißt: temp <= b; b <= a+b; a <= temp
        # (a,b) = b, a+b # funktioniert genauso
print "Ende!"

# Definition einer Funktion "Greatest Common Divider"
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Lst. 5.4: Beispiele für mehrfache Zuweisung in Python

Verhalten bei Neustart: Matlab^{TM} merkt sich Variablen beim mehrfachen Ablauf eines Skripts, auch Figures bleiben offen. Sicherheitshalber sorgt man daher für reproduzierbare Startbedingungen, indem man alle Grafiken schließt (`clf`) und alle Variablen löscht (`clear all`) - ansonsten kann es passieren, dass Skripte nach einem Neustart von Matlab^{TM} nicht mehr funktionieren, weil vorher eine im Quelltext gelöschte Variable noch im Speicher stand.

Bei Spyder bekommt man dieses Verhalten, indem man Python-Skripte mit der Option „Execute in current Python or IPython interpreter“ startet. Mit der Option „Interact with the Python interpreter after execution“ kann man auch nach dem Programmende oder einem Fehler auf die Variablen und Objekte zugreifen (z.B. im Variable Explorer).

5.4.5. Arrays, Vektoren und Matrizen

0-basierte Indizierung: Das erste Element eines Arrays (= Vektors) wird in Matlab^{TM} mit „1“ indiziert, in Python mit „0“ (zero-based indexing)!

Vektoren und Matrizen: Matlab^{TM} hat i.A. die kompaktere Schreibweise für Vektoren und Matrizen, da bei Python die entsprechende Funktionalität erst mit der NumPy Bibliothek in gültiger Python-Syntax nachgerüstet wurde. Matlab arbeitet defaultmäßig mit Matrixformaten, Scipy / NumPy Funktionen arbeiten im Allgemeinen mit Array Datentypen. `a * b` ist in Matlab daher eine Matrixmultiplikation, in NumPy eine elementweise Multiplikation. Matrixmultiplikation muss in NumPy explizit gewählt werden mit `dot(a,b)` wenn `a` und `b` Arrays sind. Man kann auch mit `a = b = np.matrix('1`

`2; 3 4')` `a` und `b` als Matrixtypen deklarieren; dann ist `a * b` eine Matrixmultiplikation und `dot(a,b)` ist eine elementweise Multiplikation - Vorsicht! Operationen zwischen Matritzen sind in Python i.A. ähnlich definiert wie in Matlab^{TM}. Siehe auch http://wiki.scipy.org/NumPy_for_Matlab_Users zu den Vor- und Nachteilen von Array und Matrix-Schreibweise in Python.

Zeilenvektoren werden in Matlab^{TM} als `[1, 2, 4]` oder `[1 2 4]` geschrieben, Python besteht auf `array([1, 2, 4])`[1]. Vektoren mit konstantem Abstand der Elemente (z.B. Zeitachse) werden in Matlab^{TM} erzeugt mit `t = [start:step:end]` oder `t = [start:end]`. Der Defaultwert für „step" ist 1, die eckigen Klammern sind optional.

In Numpy benutzt man `t = np.arange(start, end, step)`, `t = np.arange(start, end)` oder `t = np.arange(end)`. Die Defaultwerte für `start` und `step` sind 0 bzw. 1, `end` ist der erste Wert, der *nicht* im Vektor enthalten ist (Folge des zero-based indexing).

Spaltenvektoren werden in Matlab^{TM} mit `[1; 2; 3]` definiert, in Numpy wird bei eindimensionalen Arrays nicht zwischen Zeilen- und Spaltenform unterschieden, es nützt daher auch nichts die `.transpose()` Methode (bzw. das `.T` Attribut) anzuwenden.

Für die allermeisten Vektor-Operationen stört das aber nicht, für das Skalarprodukt zweier eindimensionaler Vektoren `dot(a,b)` wird z.B. `b` vor der Operation transponiert[2]. Man kann aber die Form eines Arrays auch explizit mit `a = array([1, 2, 3]); a.shape = (3,1)` oder `a = array([1,2,3]).reshape(3,1)` angeben. Alternativ kann man auch mit Matrizen arbeiten und diese dann transponieren: `a = matrix([1, 2, 3]); b = a.T` .

5.4.6. Funktionen

Allgemeine Syntax:

Parameter-Übergabe: Bei manchen Programmiersprachen werden lokale Kopien der übergebenen Parameter angelegt („Call oder Pass-by-Value"), bei anderen wird nur ein Pointer auf den Parameter übergeben („Call oder Pass-by-Reference"). Wie im vorigen Abschnitt gezeigt, nutzt Python beide Mechanismen beim Kopieren von Variablen. Das gleiche Verhalten gibt es bei der Übergabe von Parametern: Zunächst wird der Parameter als Referenz übergeben. Wird dann versucht den Parameter innerhalb der Funktion zu ändern, wird bei immutable Variablentypen eine Kopie angelegt. Bei mutable Variablen wird das Objekt selbst, also auch der Wert außerhalb der Funktion verändert! Ist das nicht erwünscht, muss eine shallow / deep copy erzwungen werden.

Variable Anzahl von Parametern: Eine variable Anzahl von Parametern kann übergeben werden, indem man ein '*' voranstellt wie das folgende Beispiel aus http://www.python-course.eu/passing_arguments.php zeigt:

```
1 def arithmetic_mean(x, *l):
2     """ The function calculates the arithmetic mean of a non-empty
3         arbitrary number of numbers """
4     sum = x
5     for i in l:
```

[1] Ohne die `array`-Anweisung würde eine Liste erzeugt, mit der man nur sehr eingeschränkt rechnen kann.
[2] Powertipp: Der Befehl `vdot(a,b)` konjugiert und transponiert automatisch `b` vor der Operation.

```
6         sum += i
7
8     return sum / (1.0 + len(l))
```

Die Parameter, die über die Reihenfolge des Aufrufs definiert sind („positional arguments“) wie `x` im obigen Beispiel müssen dabei vorangehen.

```
1 >>> arithmetic_mean(4,7,9)
2 6.666666666666667
```

Variable Anzahl von Keyword-Parametern: Ebenso kann man vorgehen bei einer variablen Anzahl von Keyword-Parametern, hier muss ein ** vorangestellt werden. Beispiel aus http://www.python-course.eu/passing_arguments.php:

```
1 >>> def f1(**args):
2 ...     print(args)
3 ...
4 >>> f1()
5 {}
6 >>> f1(de="German",en="English",fr="French")
7 {'fr': 'French', 'de': 'German', 'en': 'English'}
```

Parameter-Unpacking: Möchte man der `arithmetic_mean` Funktion z.B. eine Liste `mylist` übergeben, gibt der Aufruf `arithmetic_mean(mylist)` eine Fehlermeldung, da die Liste als Ganzes übergeben wird. Man kann die Liste aber durch Voranstellen eines * in `arithmetic\_mean(*mylist)` automatisch entpacken lassen:

```
>>> mylist = [4,7,9,45,-3.7,99]
>>> arithmetic_mean(mylist)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "statistics.py", line 8, in arithmetic_mean
    return sum / (1.0 + len(l))
TypeError: unsupported operand type(s) for /: 'list' and 'float'

>>> arithmetic_mean(*mylist)
26.71666666666667
```

Ebenso verfährt man zum automatischen Entpacken bei der Übergabe von Keyword-Parametern an eine Funktion (Voranstellen von **):

```
1 >>> def f2(a,b,x,y):
2 ...     print(a,b,x,y)
3 ...
4 >>> d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
5 >>> f2(**d)
6 ('append', 'block', 'extract', 'yes')
```

Lambda-Funktionen, Map, Filter, Reduce: -> http://www.python-course.eu/lambda.php

5.4.7. Iterationen und Generatoren

While

For:

Yield:

List und Set Comprehension: http://www.python-course.eu/list_comprehension.php Eine kompakte (und schnelle) Formulierung ... die direkt aus der Mathematik abgeleitet ist: Eine Liste der Quadratzahlen im Bereich

$$\{x^2 \mid x \in \{0, 1, \ldots, 9\}\}$$

```
y = [ (x * x) for x in range(11) ]
```

5.4.8. Klassen und Objekte

Ein Beispiel sagt mehr als 1000 Erklärungen (http://stackoverflow.com/questions/11421659/passing-variables-creating-instances-self-the-mechanics-and-usage-of-classes):

```
class Foo          (object):
    # ^class name  #^ inherits from object

    bar = "Bar" #Class attribute.

    def __init__(self):
        #        #^ The first variable is the class instance in methods.
        #        #  This is called "self" by convention, but could be any name.
        #^ double underscore (dunder) methods are usually special.  This one
        #  gets called immediately after a new instance is created.

        self.variable = "Foo" #instance attribute.
        print self.variable, self.bar  #<---self.bar references class attribute
        self.bar = " Bar is now Baz"   #<---self.bar is NOW an instance attribute
        print self.variable, self.bar

    def myMethod(self, arg1, arg2):
        # myMethod has arguments.  You would call it like this:  instance.myMethod(1,2)
        print "Arguments  in myMethod:", arg1, arg2
        print "Attributes in myMethod:", self.variable, self.bar


a=Foo() # this calls __init__ (indirectly), output:
                # >>> Foo Bar
                # >>> Foo  Bar is now Baz

print a.variable # >>> Foo
a.variable = "bar"
a.myMethod(1, 2) # output:
                # >>> Arguments  in myMethod: 1 2
                # >>> Attributes in myMethod: bar  Bar is now Baz
Foo.myMethod(a,1,2) #<--- Same as a.myMethod(1, 2).  This makes it a little more explicit
              # what the argument "self" actually is.

class Bar(object):
    def __init__(self, arg):
        self.arg = arg     # instance attribute
```

```
38         self.Foo = Foo()   # class Foo is assigned as instance attribute
39
40 b = Bar(a)  # pass instance a (class Foo) as argument to instance-attribute b.arg (class
       Bar)
41             # via Bar.__init__() - this only works at the instantiation -> b.arg = a
42 b.arg.variable = "something" # assign a new value to a.variable
43 print a.variable # >>> something
44 print b.Foo.variable # >>> Foo   - w.t.f.???
```

Ein paar Anmerkungen:

- Eine Instanz einer Klasse enthält einen Set von Daten („Attribute“); die Instanz kann auf alle Attribute aus allen Methoden zugreifen mit Hilfe von self.einDatum
- Methoden sind sehr ähnlich zu Funktionen (auch bezüglich der Parameterübergabe - siehe voriger Abschnitt). Ein wichtiger Unterschied ist, dass die aufrufende Instanz als erstes Argument übergeben wird.
- Wenn ein Attribut in einer Instanz nicht definiert ist, wird das entsprechende Attribut der Klasse als Fallback benutzt. Auf diese Art kann man Daten allen Instanzen einer Klasse (lesend) zur Verfügung stellen.
- Um Methoden der Elternklasse von einer abgeleiteten Klasse aus zu nutzen, gibt es das `parent()`-Konstrukt (?)

5.4.9. Verschiedenes

Komplexe Zahlen: Die imaginäre Einheit in Python heißt `1j`, man kann auch direkt schreiben `3.3j`. Um auf Real- und Imaginärteil einer komplexen Zahl `z` zuzugreifen, nutzt man die Attribute `z.imag` und `z.real`.

Logarithmen: Natürlicher Logarithmus `log(a)` und Zehnerlogarithmus `log10(a)` sind vordefiniert, für `log2(a)` muss man `log(a, 2)` schreiben (für beliebige Basen möglich)

Plotting: Matlab^{TM} überschreibt beim Plotten defaultmäßig alle Informationen in einer Figure (ältere Plots, Grid, Label, ...). Daher muss zuerst geplottet werden und danach die Zeichnung „dekoriert“ werden. Alternativ: Das Kommando `hold on` schaltet den Überschreibmodus ab. In Python werden Plots, Labels etc. (soweit möglich) im aktuellen Fenster überlagert, die Reihenfolge der Befehle ist daher weitgehend egal. Farben können spezifiziert werden durch Einzelbuchstaben (z.B. `'r'`) oder durch ihre RGB-Komponenten (als Tuple `(1.0, 0.5, 0.0)` in Python und als Zeilenvektor `[1.0, 0.5, 0.0]` in Matlab^{TM}). In Python können außerdem HTML-Namen (z.B. `darkslategray`) oder Hex-Codes verwendet werden (z.B. `'#ff00ff'`).

Achtung: Zum Schluss muss der Befehl `plt.show()` gegeben werden, damit die Plots überhaupt ausgegeben werden. Damit wird eine Eventloop gestartet, damit der Interpreter nicht einfach beendet und die Plots interaktiv bearbeitet werden können.

„Fehlende“ Matlab-Befehle: Für einige vertraute Matlab^{TM}-Befehle findet man anfangs kein Äquivalent:

`clear all;` Hierzu gibt es tatsächlich kein Äquivalent in Python, Löschen von Variablen ist schwierig, wird von manchen IDEs unterstützt. Im Gegensatz zu Matlab™ kann man aber immer einen neuen Interpreter starten, der quasi „bei Null“ anfängt.

`wait(2)` -> `import time; time.sleep(2)`

`find()` -> `import matplotlib.mlab as mlab; mlab.find()` oder `np.nonzero()` : gibt Indices zurück, bei denen eine bestimmte Bedingung erfüllt ist. `matplotlib.mlab` ist ein Kompatibilitätsmodul, das außerdem Ersatz für die folgenden Matlab™-Befehle liefert:

`cohere(), csd(), detrend(), griddata(), prctile(), prepca(), psd(), rk4(), specgram()`

Abb. 5.1.: Offene Klammern machen nervös ... [http://xkcd.com/859/]

5.5. Eigene Module und Packages

In Matlab™ müssen eigene Funktionen in einen File mit genau dem gleichen Namen wie die Funktion abgelegt werden. Diese Einschränkung gibt es in Python nicht - ein Python-File `myPyMod.py` mit beliebig vielen Funktionen oder Klassen `myStuff1`, `myStuff2`, ... kann als Modul importiert und jede enthaltene Funktione / Klasse verwendet werden. Jedes Modul spannt seinen eigenen globalen Namespace (s.o.) auf.

Ein Directory `myPyDir` mit Python Modulen wird als „Package“ erkannt, wenn es einen (im einfachsten Fall leeren) File mit dem Namen **`__init__.py`** enthält. Python Anweisungen in `__init__.py` werden beim Importieren ausgeführt.

Umfangreiche Pakete kann man mit weiteren Unterverzeichnissen hierarchisch strukturieren, wobei jedes Unterverzeichnis einen eigenen `__init__.py` File benötigt.

Aus dem Modul bzw. Package importiert man mit

```
1 import myStuff1 from myPyMod # bzw.
2 import myStuff2 from myPyDir.myPyMod
```

Der Name des Moduls ist dabei einfach der Filename ohne die `.py` Endung.

Beim Import eines Moduls sucht der Python Interpreter in folgender Reihenfolge:

1. aktuelles Verzeichnis
2. Verzeichnisse in der Shell Variablen `PYTHONPATH`
3. im Defaultpfad (unter *NIX meist `/usr/local/lib/python/`)

Dieser Suchpfad ist abgelegt im Systemmodul `sys` als Variable (Liste) `sys.path`. Eine übersichtliche Anzeige bekommt man mit

```
\print('\n'.join(sys.path))
```

Zusätzliche Directories können bei Bedarf mit der `append` Methode an die Liste angehängt werden:

```
import sys
sys.path.append("/home/me/mypy")
sys.path.append("../")
```

Allerdings hängt jede Ausführung (im gleichen Interpreter) die entsprechenden Directories erneut an den Pfad an.

Im folgenden wird von folgender Modulstruktur ausgegangen:

```
pyfda.py
dsp_lib.py
setup.py
pyfda_mod/
    __init__.py # from .pyfda import main
                pyfda.py
                filterbroker.py
                pyfda_lib.py
                string.py
    input_widgets/
        __init__.py # leer
        moduleX.py
        moduleY.py
    plot_widgets/
        __init__.py # leer
        moduleZ.py
```

5.5.1. Absolute Importe

Absolut hat bei Importen ähnliche Bedeutung wie bei der Navigation in Verzeichnissen, Startpunkt ist hier die eingebaute Variable `__name__`:

1. You run python pyfda.py. 2. pyfda.py does: import pyfda_mod.input_widgets.input_filter 3. input_filter.py does import pyfda_mod.plot_widgets.plot_pz

That will work as long as you have pyfda_mod in your PYTHONPATH. pyfda.py could be anywhere then.

So you write a setup.py to copy (install) the whole pyfda_mod package and subpackages to the target system's python folders, and pyfda.py to target system's script folders.

```
import pyfda_mod
import pyfda_mod.pyfda
```

http://stackoverflow.com/questions/16981921/relative-imports-in-python-3

Aufgrund des oben beschriebenen Suchmechanismusses kann ein lokales Modul / Paket z.B. ein gleichnamiges Modul aus einer Python Standardlibrary verdecken, das weiter hinten in `sys.path` steht.

Vor Python 2.5 wurde defaultmäßig zunächst im lokalen Directory des Verzeichnisses gesucht; die Anweisung `import string` in `pyfda.py` hätte daher das lokale `string` Modul gefunden und nicht das gleichnamige Python Standardmodul.

In PEP 328 wurde daher u.a. vorgeschlagen (und ab Python 2.5 implementiert) mit der Anweisung

```
1 from __future__ import absolute_import
```

absolute Importe zu priorisieren und so im o.g. Fall mit `import string` die Standardlibrary zu finden. Ab Python 3 ist dieses Verhalten Standard[3].

Man kann Untermodule

5.5.2. Relative Importe

Es gibt auch Fälle, in denen man aus dem aktuellen Verzeichnis/Modul '`.`' oder aus dem übergeordneten Verzeichnis/Modul '`..`' importieren möchte. Relative Importe sind daher auch mit dem neuen Mechanismus immer noch möglich, indem man dem Modulnamen in der Importanweisung einen zusätzlichen Punkt voranstellt:

```
1 # Importiere die Standardlibrary
2 import string
3 # Importiere pyfda.string
4 from . import string
5 # Importiere AUS pyfda.string
6 from .string import name1, name2
7 # Importiere aus übergeordnetem Verzeichnis
8 from ..dsp_lib import super_filter
9 from .. import dsp_lib
```

Aus PEP 328:

> Relative imports use a module's `__name__` attribute to determine that module's position in the package hierarchy. If the module's name does not contain any package information (e.g. it is set to `'__main__'`) then relative imports are resolved as if the module were a top level module, regardless of where the module is actually located on the file system.

Beispiele aus Python-Standardlibraries:

[3] In der Pythondoku wird fälschlicherweise behauptet, dies wäre bereits ab Python 2.7 Standard.

```
import scipy # importiere das scipy Package
a = scipy.arange(10) #  (1)
#
import scipy.signal # importiere das Unterpaket signal aus scipy
y = scipy.signal.lfilter(b, a, x) #  (2)
#
import scipy.signal.signaltools as st # importiere das Modul 'signaltools.py'
y = st.lfilter(b, a, x) # (3)
#
from scipy.signal.signaltools import lfilter # importiere die Funktion lfilter() in den
    Namespace
```

(1) `__init__.py` im `scipy` Directory enthält u.a. die Anweisung `from numpy import *`, die alle numpy-Funktionen in den Scipy-Namespace importiert. Daher steht `numpy.arange()` auch im `scipy` Namespace als `scipy.arange()` zur Verfügung.

(2) `__init__.py` im `scipy/signal` Unterverzeichnis enthält u.a. die Anweisung `from .signaltools impo` (relative Importe, s.u.). Damit stehen alle Funktionen von `signaltools.py` wie `lfilter()` im signal name scope zur Verfügung.

(3)

Wieviele Argumente beim Aufruf übergeben worden sind, kann man überprüfen mit

```
import sys
if len(sys.argv) < 2:
    print("Reads a text-file.\n\nUsage: %s filename.txt" % sys.argv[0])
    sys.exit(-1)

print(sys.argv[1])
```

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

Was nicht funktioniert:

```
# pyfda.py:
import .filterbroker as fb # Syntax error!
from . import filterbroker as fb
```

Value error: Attempted relative import in non-package

Seit Python 2.6 kann man Module relativ zum Hauptmodul referenzieren (PEP 366).

5.5.3. Ausführen von Modulen

Python Skripte, die als Module importiert werden (können), enthalten meist den folgenden Abschnitt, um das Skript unabhängig auszuführen und / oder zu testen:

```
1 if __name__ == '__main__':
2         test_my_function()
```

`__name__` ist eine eingebaute Variable, die den Wert `'__main__'` zugewiesen bekommt, wenn das Skript selbst aufgerufen wird. In diesem Fall wird der Test Code ausgeführt, andernfalls (d.h. das Modul wurde importiert) wird der Test Code ignoriert.

Achtung: Wird das Modul `myStuff` geändert, muss es neu geladen werden mit `myStuff = reload(myStuff)`!

6. Print & Plot

6.1. Matplotlib

Matplotlib ist der „Quasistandard“ für wissenschaftliche Grafik in Python, mit dem einfache 2D- und 3D-Grafiken rasch erstellt sind. Genau wie in Matlab™ muss man sich etwas Zeit nehmen, um „dekorative“ Grafiken zu erzielen, die dann aber Matlab™ mindestens ebenbürtig sind. Vor allem die Latex-Unterstützung ist in Python deutlich besser. Bei 3D - Grafiken hat Matlab™ noch klar die Nase vorn, es fehlen vor allem Optionen für Shading und Kameraeffekte. Hier kann man z.B. das Pythonmodul MayaVi einsetzen (siehe Kap. 9.7).

Zwei sehr gute Tutorials zum schnellen Einstieg in die Matplotlib mit vielen Beispielen sind
http://www.loria.fr/~rougier/teaching/matplotlib/ und
http://scipy-lectures.github.io/intro/matplotlib/matplotlib.html.

Das Untermodul **pyplot** (z.B. `import matplotlib.pyplot as plt`) stellt ein vereinfachtes (prozedurales, Matlab™-Style) Applikations Interface (API) zu den Plotting Libraries der matplotlib zur Verfügung, das z.B. beim Aufruf des `plot` Befehls automatisch `figure` und `axes` setzt. Für komplexere Optionen muss man allerdings auf das objekt-orientierte API zurückgreifen (siehe Listing 6.1 und 6.2). Um auf das Axes-Objekt zugreifen zu können, muss hier immer mit `add_subplot(xxx)` mindestens ein Subplot instanziiert werden.

```
1 import matplotlib.pyplot as plt
2 from numpy import arange
3 plt.figure()  # optional
4 %
5 plt.plot(arange(9),arange(9)**2)
6 plt.show()
```

Lst. 6.1: Prozedurales API der Matplotlib

```
1 import matplotlib.pyplot as plt
2 from numpy import arange
3 fig = plt.figure()
4 ax = fig.add_subplot(111)
5 ax.plot(arange(9),arange(9)**2)
6 plt.show()
```

Lst. 6.2: Objekt-orientiertes API der Matplotlib

Das Modul **pylab** geht noch einen Schritt weiter und wirft `numpy` und `matplotlib.pyplot` in den gemeinsamen Namespace, so dass man sofort z.B. `x = linspace(0, 2*pi); plot(x,sin(x)` oder `hist(randn(500))` eingeben kann. `pylab` sollte nur im interaktiven Modus (z.B. innerhalb von IPython), aber nicht in eigenen Skripten verwendet werden.

Siehe auch „Using matplotlib in a python shell“, http://matplotlib.org/users/shell.html#using-matplotlib-in-a-python-shell

6.1.1. Fonts und Formeln

Welche Fonts von der Matplotlib verwendet werden, wird in der .matplotlibrc festgelegt (s.u.). Plots können mit Formeln und Sonderzeichen sehr schön beschriftet werden, da die Matplotlib einen eigenen TEX-Parser („mathtext“) und Renderer enthält, der eine Untermenge der TeX- sowie eigene Fonts. Wenn LATEX auf dem Computer installiert ist, kann die Beschriftung auch komplett mit LATEX gesetzt werden.

Ein kniffliges Thema ist dabei die Mischung von Text und Formelzeichen, da beide jeweils mit eigenem Renderer und eigenen Fonts gesetzt werden. Unterschiedliche Fonts und Zeichengrößen sind oft die Folge. Das Problem kann man auf die folgenden Arten lösen:

LaTeX

Diese Variante liefert das beste Schriftbild, ist allerdings auch am langsamsten und fehleranfälligsten (LATEX-Distribution und Ghostscript müssen erkannt werden etc.). Z.Z. funktioniert sie nur mit den Backends `agg`, `PS` und `PDF`. Details und Tutorial siehe http://matplotlib.org/users/usetex.html. Achtung: Auch Text außerhalb von `$ ... $` wird jetzt nach TEX-Regeln gesetzt; Zeichen wie `$, #, %` können daher unerwartete Ergebnisse bringen!

In der `matplotlibrc` muss hierzu `text.usetex: True` gesetzt sein (s.u.) bzw. im Code aktiviert werden mit

```
from matplotlib import rc
rc('text', usetex=True)
```

Beschriftung mit MathText und normalem Text

Genau wie in LATEX müssen Zeichen, die im Math-Modus gesetzt werden sollen, zwischen `$ ... $` stehen. Da `\t` als Tabulator und `\n` als Zeilenumbruch interpretiert werden, sollten Strings mit `r' ... '` als raw deklariert werden, also z.B.

```
ylabel(r'Group delay $\tau_g \; \rightarrow$')
title(r'$|H(\omega, t)|$ bei $t=%0.1f$ s' %(bins[100]))
```

Die zweite Zeile zeigt, dass auch Formatierungsoptionen in den MathText-String geschrieben werden dürfen (das geht mit Latex nicht). Eine Font-Kombination, die relativ gut aussieht, ist in den Defaultoptionen weiter unten beschrieben.

Beschriftung mit MathText

Wenn man relativ viele Formeln und wenig Text hat, kann man einen einheitlichen Font erreichen, indem man den Text in den TeX-Teil schreibt mit z.B.

```
ylabel(r'$\textrm{Group\; delay} \tau_g \; \rightarrow$')
```

Leerzeichen muss man jetzt auch im Text mit `\,` (schmal) oder `\;` erzwingen, da im Math-Modus Leerzeichen normalerweise elimiert werden. Das Aussehen des Texts kann man u.a. mit den TEX-Befehlen `\textrm{Serif},\textit{kursiv}, \textbf{fett}` beeinflussen.

Mehr Infos und ein Tutorial gibt es unter http://matplotlib.org/users/mathtext.html.

6.1.2. Anpassungen - `matplotlibrc` usw.

Die Defaultoptionen für Linienstärke, Font, Farben, ... der Plots werden eingestellt im File `matplotlibrc`. Dieses File ist in Python Syntax und wird in der folgenden Reihenfolge gesucht (siehe auch http://matplotlib.org/users/customizing.html):

1. Im aktuellen **Working Directory**: Dort werden typischerweise projektspezifische Einstellungen abgelegt
2. Im userspezifischen **matplotlib Directory**:
   Unter *nix ist dies meist `.config/matplotlib/matplotlibrc`,
   unter Windows `Users and Documents/der_user/.matplotlib`. Den genauen Ort erfährt man mit `matplotlib.get_configdir()`, das Verzeichnis lässt sich bei Bedarf mit der Umgebungsvariablen `MPLCONFIGDIR` ändern, im Code z.B. mit
   `os.environ['MPLCONFIGDIR']='C:\\users\\muenker\\.matplotlib'`.
   (Das hat aber keine Auswirkung?!) Bei Winpython wird anscheinend das Windows-Userdirectory automatisch ermittelt?
3. Im **`$MATPLOTLIBRC`** Directory (erst ab Matplotlib 1.5 ?)
4. Im **Installationsdirectory** `MPLIB`, das man sich mit `matplotlib.__file__` anzeigen lassen kann, finden sich die Defaulteinstellungen `MPLIB/mpl-data/matplotlibrc`, unter Windows ist das z.B.
   `C:\Python27\Lib\site-packages\matplotlib\mpl-data\matplotlibrc`. Änderungen an diesem File werden beim nächsten Update / Neuinstallation überschrieben und sollten daher unbedingt gesichert werden! Hier sind bereits alle Konfigurationsmöglichkeiten (auskommentiert) enthalten, man kann sich also leicht diejenigen herauspicken, die man ändern will und die Änderungen z.B. an das Ende des Files stellen

Achtung: Sobald *ein* `matplotlibrc` File gefunden wurde, wird die Suche abgebrochen (fallback, kein cascading). `matplotlib.matplotlib_fname()` gibt aus, welcher Konfigurationsfile verwendet wird.

Mit `plt.style.use` (siehe Kap. 6.1.3) kann man mehrere `matplotlibrc` Files kaskadiert einbinden

```
#========== CHANGES to matplotlibrc ================================
### FONT
font.family       : serif  # Default: sans-serif
font.size         : 16.0 # Default 12.0
font.serif        : Times, Times New Roman, Palatino, Computer Modern Roman
font.sans-serif   : Helvetica, Avant Garde, Computer Modern Sans serif
font.cursive      : Zapf Chancery
font.monospace    : Courier, Computer Modern Typewriter

#text.usetex      : True # activate LaTeX rendering for everything (Default: False)

```

```
12 ### AXES
13 axes.grid         : True   # display grid # Default: False
14 axes.color_cycle  : r, b, g, c, m, y, k  # color cycle for plot lines
15                                          # as list of string colorspecs:
16                                          # single letter, long name, or
17                                          # web-style hex
18
19 #grid.color        : darkgrey   # grid color, Default: black
20
21 axes.xmargin      : 0.0  # relative x margin with autoscale (Default: 0)
22 axes.ymargin      : 0.01 # relative y margin with autoscale (Default: 0)
23
24 axes.formatter.limits : -3, 3 # use scientific notation if log10
25                               # of the axis range is smaller than the
26                               # first or larger than the second # was: -7, 7
27 axes.formatter.use_mathtext : True # When True, use mathtext for scientific
28                                    # notation of multiplier # was: False
29
30 lines.color       : r    # no effect on plot() and stem(); see axes.color_cycle
31 lines.markersize  : 8    # markersize in points; Default 6
32 lines.linewidth   : 1.5  # line width in points; Default 1.0
33
34 legend.scatterpoints : 1 # number of scatter points in legend (Default: 3)
35
36 savefig.dpi         : 300      # figure dots per inch # was 100
37 #savefig.facecolor   : white    # figure facecolor when saving
38 #savefig.edgecolor   : white    # figure edgecolor when saving
39 #savefig.format      : png      # png, ps, pdf, svg
40 savefig.bbox        : tight   # 'tight' or 'standard' # was: 'standard'
41 savefig.pad_inches  : 0.02   # Padding to be used when bbox is set to 'tight' # was: 0.1
42 savefig.directory   :         # default directory in savefig dialog box,
43
```

Matlab™ hat übrigens einen ähnlichen Mechanismus (http://blogs.mathworks.com/community/2009/12/07/the-preferences-directory/), das Kommando `prefdir` gibt das Directory mit den Default settings zurück. Ein Teil der Plot-Eigenschaften lassen sich auch über das Menu mit „Preferences“ einstellen.

6.1.3. Styles

Einstellungen in der gleichen Syntax wie im matplotlibrc File lassen sich auch in Style-Files `*.mplstyle` abspeichern und dann hierarchisch einbinden:

Im Pfad: Mit Suffix `*.mplstyle` und optionaler Pfadangabe

Mpl-Directories: Ohne Suffix, in Subdirectories `stylelib` der oben angebenen Matplotlib-Settings directories

```
import matplotlib.pyplot as plt
print(plt.style.available) # alle gefundenen Stylefiles
plt.style.use('my_new_style') # sucht in stylelib Directories
plt.style.use('my_new_style.mplstyle') # sucht im Pfad
```

Lst. 6.3: Matplotlib-Stylefiles

https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/mpl-data/stylelib/

6.1.4. XKCD-style Plots

Wenn Mathematik mal nicht so steril aussehen soll, kann man Plots einen „händischen“ Look (Abb. 6.1) geben, nachempfunden den XKCD - Cartoons (http://xkcd.org).

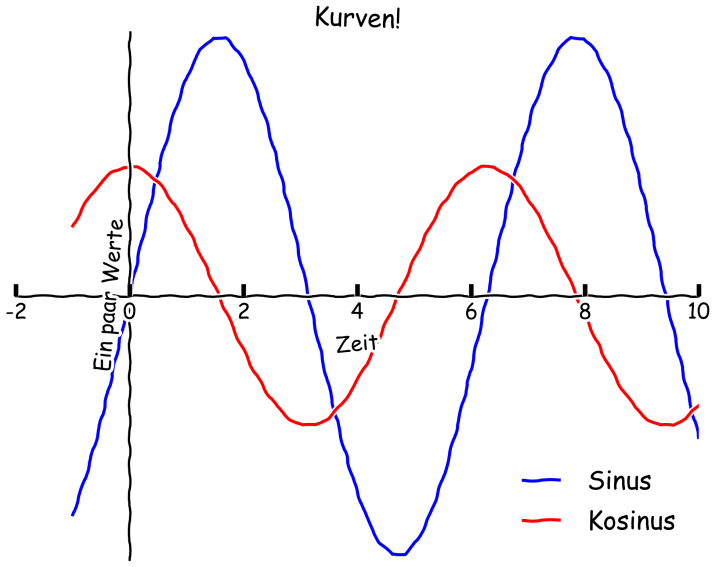


Abb. 6.1.: XKCD Style Plotting

XCKD-style Plots sind seit Version 1.3.0 der Matplotlib ohne weitere Importe möglich.

```
import matplotlib.pyplot as plt
from numpy import linspace, sin, cos
plt.xkcd(scale=2, length=200, randomness=10) # Aktiviere XKCD - Modus
fig = plt.figure(1)
ax = fig.add_subplot(1, 1, 1)
ax.spines['right'].set_visible(False)
```

```
ax.spines['left'].set_position('zero')
ax.spines['top'].set_visible(False)
ax.spines['bottom'].set_position('zero')
ax.xaxis.set_ticks_position('bottom')
plt.yticks([])
t = linspace(-1, 10)
plt.plot(t, sin(t), label = 'Sinus')
plt.plot(t, 0.5* cos(t), label = 'Kosinus')
plt.xlabel('Zeit', rotation=5); plt.ylabel('Ein paar Werte',rotation=85)
plt.title('Kurven!', rotation=-2)
plt.legend(loc='best', frameon=False)
plt.show()
```

Lst. 6.4: Beispiel für XKCD-Style Plots mit Matplotlib

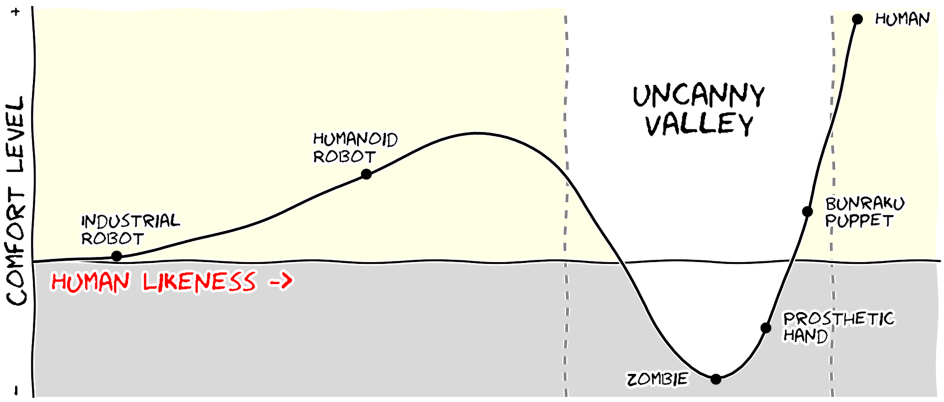


Abb. 6.2.: Ein aufwändigeres XKCD-Beispiel (Listing E.7) [doi:10.1371/journal.pcbi.1003833.g008]

Weitere Beispiele unter
http://jakevdp.github.io/blog/2013/07/10/XKCD-plots-in-matplotlib/ und
http://matplotlib.org/xkcd/gallery.html.

6.2. Alternativen und Add-Ons zur Matplotlib

Trotz ihrer Leistungsfähigkeit hat die Matplotlib ein paar Schwächen, die verschiedene Grafikmodule ausbessern:

6.2.1. Schöner plotten - ggplot und seaborn

Die voreingestellten Defaults der Matplotlib liefern schon ansehlichere Resultate als beispielweise Matlab und lassen sich leicht an den eigenen Geschmack anpassen (s.o.). Trotzdem kann man hier noch einiges herausholen (Abb. 6.3). Wie man in Listing E.6 sehen kann, kann der Aufwand beträchtlich sein. **ggplot** (http://ggplot.yhathq.com/) und **Seaborn** (http://www.stanford.edu/~mwaskom/software/seaborn/index.html) sind Add-Ons zur Matplotlib und zielen darauf ab, professionell aussehende Grafiken mit minimalem Aufwand (= sinnvolle Defaults, übersichtliche API) zu erstellen. Hauptzielrichtung ist die Visualisierung von Statistikdaten. Listing 6.5 und Abb. 6.4 aus der Seaborn-Gallery zeigen wie einfach man attraktive Graphen erstellen kann.

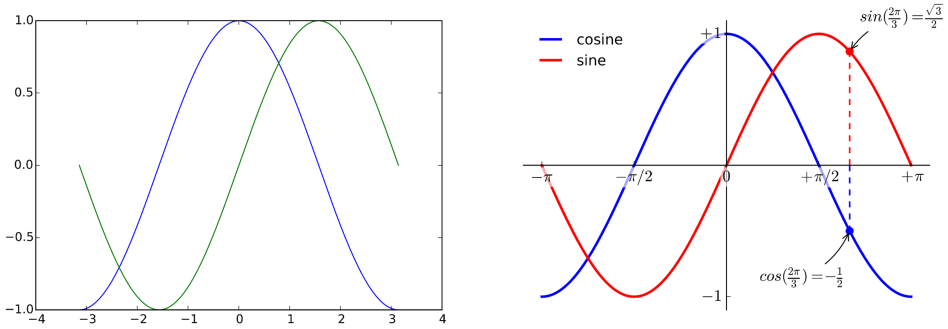


Abb. 6.3.: Mehr herausholen aus der Matplotlib (Listing E.5 und Listing E.6)
[doi:10.1371/journal.pcbi.1003833.g004]

`ggplot` basiert auf der Syntax von `ggplot2` (http://ggplot2.org/), einem Plotting System für die Statistiksprache `R`, das sich gut mit `pandas` Datentypen versteht. Auch komplexe Grafiken mit vielen Ebenen lassen sich leicht erstellen.

6.2.2. Webbasierte Plots - Bokeh, mpld3 und Plotly

Es ist einfach, mit Python Seiten für Browser zu generieren oder Webserver aufzubauen, entsprechende Funktionalität ist mit dem `WebAgg` Backend erst seit Version in Matplotlib integriert. **`Bokeh`**, **`mpld3`** und **`Plotly`** erzeugen webbasierte, interaktive Grafiken. Aus den zu visualisierenden Daten wird durch ein Modul mit Python-API ein JSON-Objekt generiert, die eigentliche Visualisierung erledigt dann eine Javascript-Bibliothek im Browser. Über Matplotlib lassen sich auch `seaborn` oder `ggplot` Plots ins Web bringen.

```
import numpy as np
import pandas as pd
import seaborn as sns

# create a 2D random distribution
# with some covariance
rs = np.random.RandomState(None)
mean = [0, 0] # mean values
cov = [(1, .5), (.5, 1)] # cov. matrix
x1, x2 = rs.multivariate_normal(mean, cov,
     500).T
x1 = pd.Series(x1, name="$X_1$")
x2 = pd.Series(x2, name="$X_2$")

# select style of seaborn plot:
sns.set(style="darkgrid")
# create plot
g = sns.jointplot(x1, x2, kind="kde",
                  size=7, space=0)
```

Lst. 6.5: Beispiel für Seaborn-Plot ...

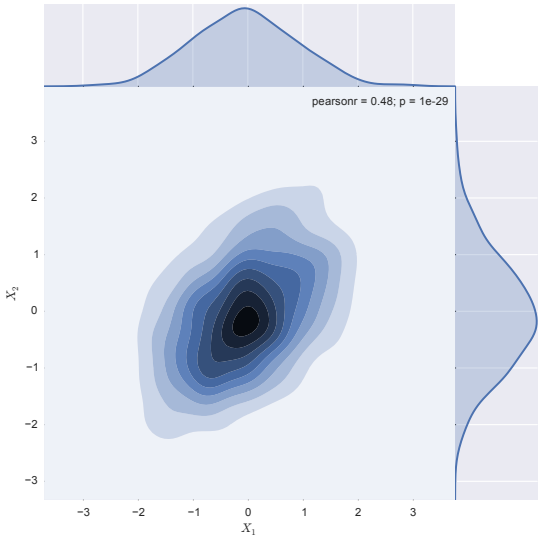


Abb. 6.4.: ... von zweidimensionalen Daten

`mpld3` (http://mpld3.github.io/) legt den Schwerpunkt auf Kompatibilität zur Matplotlib (muss installiert sein) und setzt die weitverbreitete Javascript-Bibliothek `d3.js` (http://d3js.org/) ein.

`Bokeh` (http://bokeh.pydata.org/) verwendet die `BokehJS` Library und ermöglicht die Visualisierung großer Datenmengen oder von Streaming Data, der Python Client kommt ohne Matplotlib aus und läuft daher auch z.B. auf einem Raspberry Pi oder einem Beagle Bone. Wenn die Matplotlib vorhanden ist, können mit `mpl.to_bokeh()` aber auch viele Matplotlib-Features genutzt werden. Bokeh wird als Open Source Projekt auf GitHub von Continuum Analytics (Anaconda Distribution) unterstützt, eine interaktive Gallerie gibt es unter

`Plotly` (https://plot.ly/) geht noch einen Schritt weiter - hier gibt es APIs nicht nur für Python sondern auch für Matlab, R, Excel, Arduino (C) und Raspberry Pi (Python). Die Daten können importiert und exportiert werden und ermöglichen so kollaboratives Arbeiten mit unterschiedlicher Software. Außerdem können die Plots in verschiedenen Vektor- und Bitmap-Formaten exportiert werden. Allerdings läuft die JS-Software nur auf den Servern des Anbieters, man benötigt einen kostenfreien Account. Für Firmen gibt es auch Enterprise-Angebote (kostenpflichtig).

http://www.instructables.com/id/Plotly-Atlas-Scientific-Graph-Real-Time-Dissolved-/

6.2.3. Hardwarebeschleunigung - VisPy und PyQtPlot

Die fehlende Hardware-Beschleunigung lässt die Plot-Geschwindigkeit bei großen Datenmengen oder bei 3D-Grafiken in die Knie gehen. **`VisPy`** (http://vispy.org/) ist ein vielversprechendes Projekt, bei dem sich die Entwickler von vier ähnlichen Vorgängerprojekten zusammengetan haben, um über eine Python-API auf die OpenGL-Library zuzugreifen.

`pyQtGraph` (http://www.pyqtgraph.org/) setzt auf numpy und PyQt4 und OpenGL Hardwarebeschleunigung.

6.2.4. Dies und das

http://zulko.github.io/blog/2014/09/20/vector-animations-with-python/ zeigt beeindruckende Animationen, generiert mit Hilfe der Bibliotheken Gizeh (https://github.com/Zulko/gizeh) und MoviePy (http://zulko.github.io/moviepy/)

6.3. Schöner Drucken

Ähnlich wie mit `sprintf()` in C, lassen sich auch in Python Druckausgaben vielfältig formatieren (http://docs.python.org/2/library/stdtypes.html#string-formatting).

Typ	Erklärung	Beispiel
'd'	Signed integer decimal	%3d
'i'	Signed integer decimal	
'o'	Signed octal value.	
'u'	Veraltet – identisch mit 'd'	
'x', 'X'	Signed hexadecimal	
'e', 'E'	Floating Point Exponentialformat	%9.2e
'f', 'F'	Floating Point Dezimalformat, 'f' druckt den Dezimalpunkt nur wenn er benötigt wird.	
'g', 'G'	Automatisches Floating Point Format: Exponenzialformat, wenn der Exponent kleiner als -4 oder als die Anzahl der verlangten Nachkommastellen ist, ansonsten Dezimalformat	
'c'	Single character (accepts integer or single character string)	
'r'	String (converts any Python object using repr())	
's'	String (converts any Python object using str())	
'%'	Keine Umwandlung, erzeugt '%' Zeichen im Ergebnis	

- Ob der Typcode groß oder klein angegeben wird, entscheidet über Formatierungsdetails, wie z.B. ob das „e“ im Exponenzialformat groß oder klein geschrieben wird
- Die Formatierungsanweisungen können mit `\n` (Zeilenumbruch) und `t` (Tabulator) kombiniert werden

Die Formatierungsanweisungen funktionieren bei allen drei Arten zu drucken ähnlich: (a) alte Printanweisung, (b) neue Printfunktion (Python > 2.6) und (c) neue Formatanweisung in Python 3:
(a) `print "pi = %f, 3.3 uF = %9.2E F" %(pi,3.3e-6)`
(b) `print ("pi = %f, 3.3 uF = %9.2E F" %(pi,3.3e-6))`
(c) `print ("pi = {0:f}, 3.3 uF = {1:9.2E} F".format(pi,3.3e-6))`

Alle drei Anweisungen erzeugen das gleiche Ergebnis, nämlich
`pi = 3.141593, 3.3 uF =  3.30E-06 F`

Bei der neuen Formatanweisung der Python 3 Print-Funktion sind geschweifte Klammern Platzhalter für Zahlen oder Strings, die mit der `str.format` Methode durch die übergebenen Objekte ersetzt werden. Das erste Argument im Platzhalter ist dabei die Position des Objekts, durch das er ersetzt werden soll. Das zweite Argument legt die eigentliche Formatierung fest (hier: 9 Stellen insgesamt, floating point, 2 Nachkommastellen). Da die Vorzüge der neuen Formatierung viele nicht so recht überzeugen, wird wohl auch in die alte Formatierung noch länger erhalten bleiben.

7. IPython Notebook und andere wolkige Themen

IPython („Interactive Python“) ist eine Konsole für Python, die im Vergleich zu einer „normalen“ Konsole wesentlich interaktiver bedient werden kann, besseren Zugriff auf Betriebssystemfunktionen bietet (z.B. mit `myfiles = !ls`) und Inline-Plots darstellen kann. Übersichtliche Darstellung von Fehlern, „Magic functions“ wie z.B. `%timeit, %debug, %history` etc., Tab Completion und einfachen Zugriff auf die eingebaute Hilfe durch Anhängen von `?` an ein Objekt runden die Funktionalität ab. Man kann IPython benutzen wie eine ganz normale Konsole, aber dann verpasst man ziemlich viel ...

Achtung: Für IPython muss u.U. eine Ausnahmeregel zur (Windows-)Firewall hinzugefügt werden, da immer ein localhost Server gestartet wird - egal, ob IPython als Konsole oder im Browser verwendet wird.

7.1. IPython als interaktive Shell

Man sollte sich zumindest unter `http://ipython.org/ipython-doc/stable/interactive/tutorial.html` mit den absoluten Basics vertraut machen, ein kürzeres oder längeres Video unter `http://ipython.org/videos.html` oder eine Präsentation unter `http://ipython.org/presentation.html` anschauen.

Zum interaktiven Arbeiten wird IPython normalerweise mit dem `-pylab` Switch gestartet, mit dem das `pylab` Modul gleich beim Start geladen wird. Pylab lädt `numpy` und `matplotlib.pyplot` in den gemeinsamen Namespace kopiert, so dass man z.B. mit `x = linspace(0, 2*pi); plot(x,sin(x))` sofort einen Plot innerhalb der Konsole oder in einem eigenen Fenster erhält (je nach Voreinstellung). Nicht schlecht, oder?

7.1.1. Inline vs. interactive Plots

IPython unterstützt Plots in eigenem Fenster, die man exportieren kann, in die man hineinzoomen kann etc. Beim Aufruf mit dem Switch `--inline` werden Inline-Plots erzeugt, d.h. `plot(x,y)` zeichnet ein Fensterchen in die interaktive Konsole, das nicht weiter manipuliert werden kann. In Spyder lässt sich der Modus *vor* dem Start einer neuen Konsole wählen (-> Tools -> Preferences -> IPython console -> Backend -> Inline bzw. Qt).

7.2. Python im Browser - das IPython Notebook

Seine Stärken kann IPython aber erst so richtig ausspielen, wenn es über ein Websocket Interface im Webbrowser (lokal oder remote) als „Notebook“ ausgeführt wird.

Hier kann man Python Code anreichern durch Grafiken, durch Text im Markdown- oder im Latex-Format (inkl. Formeln, die mit MathJax gerendert werden) (Abb. 7.1). Der Python Code „lebt“, d.h. er kann geändert und immer wieder zellenweise oder komplett ausgeführt werden. Mit Notebooks lässt sich daher prima experimentieren und Code stückchenweise entwickeln und gleichzeitig dokumentieren, daher sind sie auch ideal für die Lehre geeignet. Notebooks könnnen einfach im interaktiven (sehr kompakten) *.ipynb - Format weitergegeben werden, aber auch als statisches HTML oder als PDF.

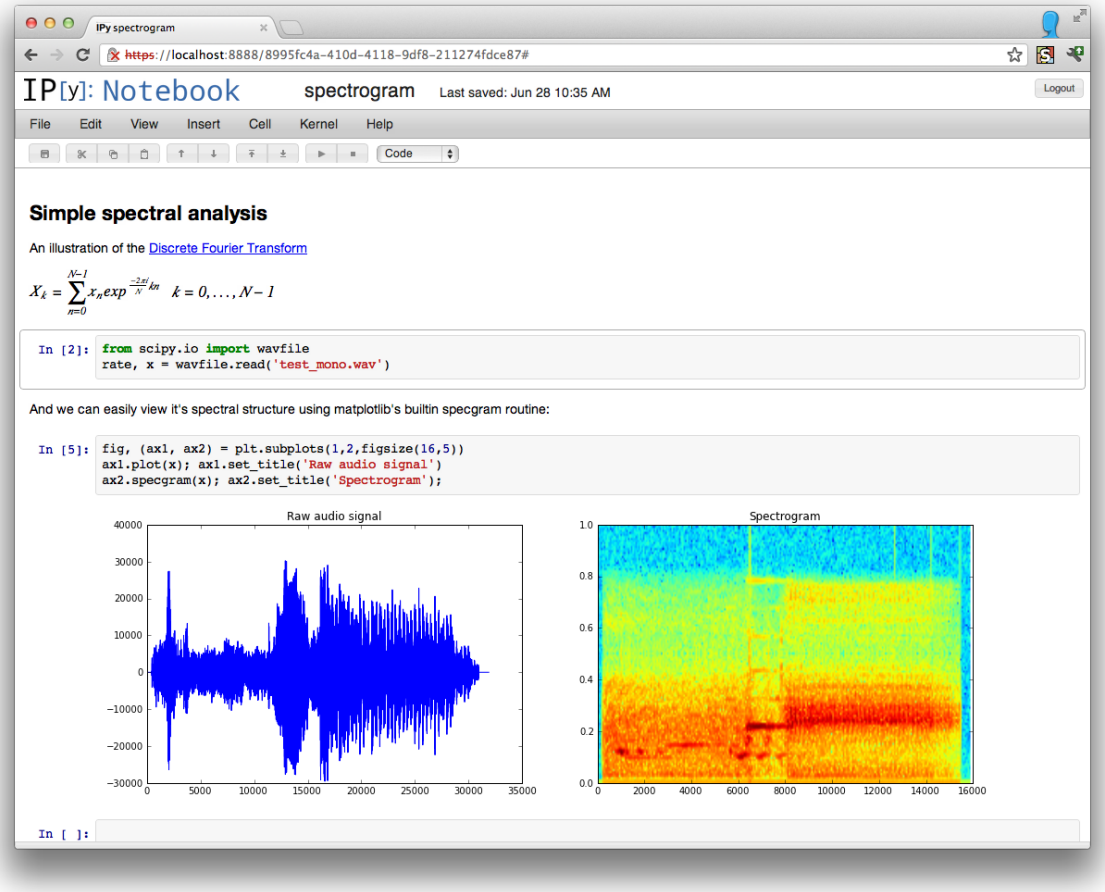


Abb. 7.1.: Screenshot eines IPython Notebooks [http://ipython.org]

Das IPython Notebook ist eine Anwendung, die auf zwei Prozesse aufgeteilt ist: Das grafische „Frontend“ ist in Javascript programmiert und läuft im Web Browser. Alternativ kann das Frontend auch die IPython - Konsole sein wie oben beschrieben.

Das „Backend“, auf dem der Python Code liegt und ausgeführt wird, kann dabei der eigene PC sein, eine virtuelle Maschine im Intranet oder in der „Cloud“.

Aufgrund dieser Zweiteilung muss man eine Ausnahmeregel für die Firewall definieren (selbst wenn man nur lokal arbeitet), sonst funktioniert leider gar nichts.

Sage Notebooks verfolgen einen ähnlichen Ansatz, allerdings ist SAGE eine abgeschlossene Umgebung mit eigenem Packaging System. Das hat als Konsequenz:

- Um mit SAGE unter Windows zu arbeiten, muss dort Linux in einer virtuellen Maschine laufen. Eine Alternative könnte Wubi sein (http://wiki.sagemath.org/WubiGuide), ein Ubuntu Linux Installer, der Ubuntu auf einer Windows - Partition als großen File installiert.
- Eine bestimmte Version von SAGE setzt auf einer bestimmten Python-Version auf, die nicht geändert werden kann.

Bei Sage wird eine Abstraktionschicht zwischen Filesystem und Interpreter eingezogen, während IPython direkten Zugriff auf lokale Daten und Skripte erlaubt (http://mail.scipy.org/pipermail/ipython-user/2012-January/009021.html). Für eine lokale Installation ist IPython sehr praktisch, auf einem remote Server kann IPython zum Albtraum des Admins werden - hier ist u.U. Sage die bessere Alternative, es setzt auf einem Apache Server auf.

IPython wird seit mehr als 10 Jahren entwickelt und gehört inzwischen zum „Tafelsilber“ der Python-Community. Seit Anfang 2013 werden 7 Entwickler von der UC Berkeley und Cal Poly San Luis Obispo (Partner-Hochschule der HM!) für 2 Jahre aus einer öffentlichen Stiftung bezahlt mit dem Ziel, das IPython Notebook als allgemeines Werkzeug weiterzuentwickeln für offenes, kollaboratives und reproduzierbares Scientific und Technical Computing.

Notebooks sind

Zellbasiert: Das Benutzerinterface basiert auf „Zellen“: Jede Notebook-Zelle kann einfachen Text, Rich Text (Markdown Syntax) oder Python Code enthalten. Man kann die Zellen einzeln editieren oder ausführen oder man kann das ganze Notebook auf einmal ablaufen lassen. Plots werden statisch im Browser dargestellt, können also leider (noch) nicht gezoomt o.ä. werden.

Einen Vorteil des zellbasierten Interfaces übersieht man leicht: In einem normalen Terminal gibt jeder Fehler im Code einen langen Stack Trace aus, der den restlichen Code vom Bildschirm schiebt. Im IPython Notebook kann man jede Zelle einzeln („Run Cell“) ausführen lassen; Fehlermeldungen werden übersichtlich unterhalb der Zelle ausgegeben. Den Fehler korrigieren und den nächsten Versuch starten geht damit ratz-fatz.

Webbasiert: Speziell für die Lehre ist es sehr praktisch, wenn alle Studierenden sich auf einem vorkonfigurierten Server einloggen können: Niemand muss mehr selbst etwas installieren - und alle arbeiten mit der gleichen Version arbeiten.

Textbasiert: IPython Notebooks werden im JSON - Format[1] mit der Endung `*.ipynb` als *single File* abgespeichert und können so einfach weitergegeben werden. Aufgrund des Textformats können Notebooks auch zusammen mit Versionierungstools eingesetzt werden.

```
{
 "metadata": {
  "name": "Part 1 - Running Code"
 },
 "nbformat": 3,
 "nbformat_minor": 0,
 "worksheets": [
```

[1] Java Script Object Notation, ein kompaktes Format zum Austausch von Daten zwischen Anwendungen, das von Menschen und Computern gelesen werden kann, ähnlich XML.

```
{
 "cells": [
  {
   "cell_type": "heading",
   "level": 1,
   "metadata": {},
   "source": [
    "Running Code in the IPython Notebook"
                        ...
```

Lst. 7.1: Beispiel für JSON-Quelltext eines IPython-Notebooks

Zwei enthusiastische Kritiken von IPython Notebooks in „Living in an Ivory Basement - Stochastic thoughts on science, testing, and programming“ (http://ivory.idyll.org/blog/teaching-with-ipynb-2.html) und in http://wakelift.de/posts/ipynb/

Es gibt ein paar prinzipbedingte **Nachteile:**

Nicht-lineare Codeverarbeitung: Da Codezellen in einem Notebook einzeln verändert und ausgeführt werden können, werden Zellen „downstream“ nicht automatisch aktualisiert. Durch „run all“ kann man allerdings alle Zellen ausführen und aktualisieren.

Long-Running Jobs: Umfangreiche Skripte mit langer Laufzeit sollte man lieber nicht auf einem Remoteserver, sondern von der Kommandozeile aus laufen lassen, da IPython eine stabile Internetverbindung erwartet. Das ist eine der Stellen, an denen IPython gerade überarbeitet wird.

Große Projekte: IPython ist für interaktives Arbeiten und Experimentieren gedacht, nicht für die Entwicklung großer Projekte.

Demgegenüber stehen viele **Vorteile:**

Einfach zu benutzen: Aufsetzend auf einer bestehenden Python-Installation oder noch besser (für die Lehre) beim Arbeiten auf einem Remote-Server kann man / Studi sofort loslegen mit Programmieren.

Demos / Unterricht: Im Notebook kann Code während der Vorlesung / Präsentation entwickelt und anschließend als interaktives Notebook oder in einem statischen Format weitergegeben werden.

Interaktives Arbeiten: Die Mischung aus Text, Grafiken, „Taschenrechner“ und Live-Code kann man auch als interaktives Laborbuch zum Experimentieren und Dokumentieren verwenden

Notebook Viewer: Mit Hilfe von http://nbviewer.ipython.org kann man sich interaktive *.ipynb Notebooks aus dem Netz auch ohne Python Installation als statische HTML Notebooks anzeigen lassen

Exportformate: Aus Notebooks können leicht PDF - Dokumente als Skript erzeugt werden; im Unterricht oder zum Experimentieren zu Hause werden die Notebooks zum Leben erweckt.

Eigene Module: Eigene lokale Skripte und Module können importiert werden; IPython ist keine „Sandbox“, die einen davon abhält die vollen Möglichkeiten von Python auszuschöpfen. Mit Hilfe des Moduls `serial` sollte z.B. auch der Zugriff auf lokale Hardware problemlos möglich sein. Trotzdem können Einsteiger mit geringem Aufwand Python ausprobieren.

7.2.1. Arbeiten mit vorgefertigten Notebooks

Ein IPython Notebook startet man mit `ipython notebook --pylab inline` von der Kommandozeile aus. Mit der Winpython- und der Anaconda-Distribution hat man es noch einfacher, hier gibt es ein Skript / Icon, mit dem man direkt ein IPython Notebook starten kann.

Die Option `pylab` sorgt dafür, dass beim Start bereits numpy, scipy und matplotlib geladen und im interaktiven Modus nicht explizit importiert werden müssen, mit der Option `inline` werden Plots im Browser und nicht in eigenen Fenstern dargestellt.

Im Default-Browser öffnet sich nun das IPython Dashboard mit einer localhost URL (`http://127.0.0.1:8888/` oder ähnlich) in einem neuem Tab. In Abb. 7.2 wurde gerade das Notebook „Sampling Theorem“ geöffnet. Mit **upload** wird das Notebook zum Server übertragen (auch wenn der hier auf dem gleichen Rechner läuft) und kann danach durch Anklicken gestartet werden.

Nach dem Start präsentiert sich ein Notebook wie in Abb. 7.1, die Grundbedienung ist fast selbsterklärend:

Notebooks sind zellenweise organisiert; eine Zelle kann entweder Code („live“) enthalten, eine Überschrift oder „Markdown“, d.h. Rich Text (`http://de.wikipedia.org/wiki/Markdown`). Einfache Formatierungen sind z.B. *kursiv* oder **fett**. Markdown Zellen können auch Inline-Latex-Formeln enthalten, umgeben von `$ ... $` wie üblich oder von `$$ ... $$` für `\mathmode` Formatierungen. Mit einem Doppelklick auf eine Zelle gelangt man in den Editiermodus, der durch einen Klick daneben beendet wird. **Achtung:** Es gibt noch kein Autosave!!

Code kann zellenweise ausgeführt werden mit Cell -> Run oder <**SHIFT**>-<**ENTER**>; Variablen aus vorhergehenden Zellen kann man initialisieren, indem man mit Cell -> Run All zunächst das gesamte Notebook ausführt.

Unter `https://github.com/ipython/ipython-in-depth` finden Sie zahlreiche Anleitungen / Beispiele, zum Einstieg empfohlen „Running Code in the IPython Notebook“ (`runningcode.ipynb`)

Der freie Webservice (`http://nbviewer.ipython.org`) implementiert einen **Notebookviewer**, mit dem man sich öffentlich zugängliche IPython Notebooks im Webbrowser als statisches HTML ansehen (aber nicht interaktiv bearbeiten) kann, indem man die URL in den Notebookviewer kopiert. Auf der Startseite finden sich auch viele Beispiele für Notebooks.

Mit Hilfe des nbviewers kann man auch öffentliche oder eigene Notebooks zu statischen HTML, PDF o.ä. Files umwandeln. Zuerst macht man das Notebook per Dropbox, Github o.ä. im Web verfügbar und gibt dann die öffentliche URL im Notebookviewer ein (siehe `http://nbipython.blogspot.de/2012/11/get-started.html`). Seit IPython Version 1.0 ist dies auch direkt aus dem Notebook heraus möglich, seit 2015 kann auch GitHub dort abgelegte Notebooks direkt als (statische) Notebooks rendern.

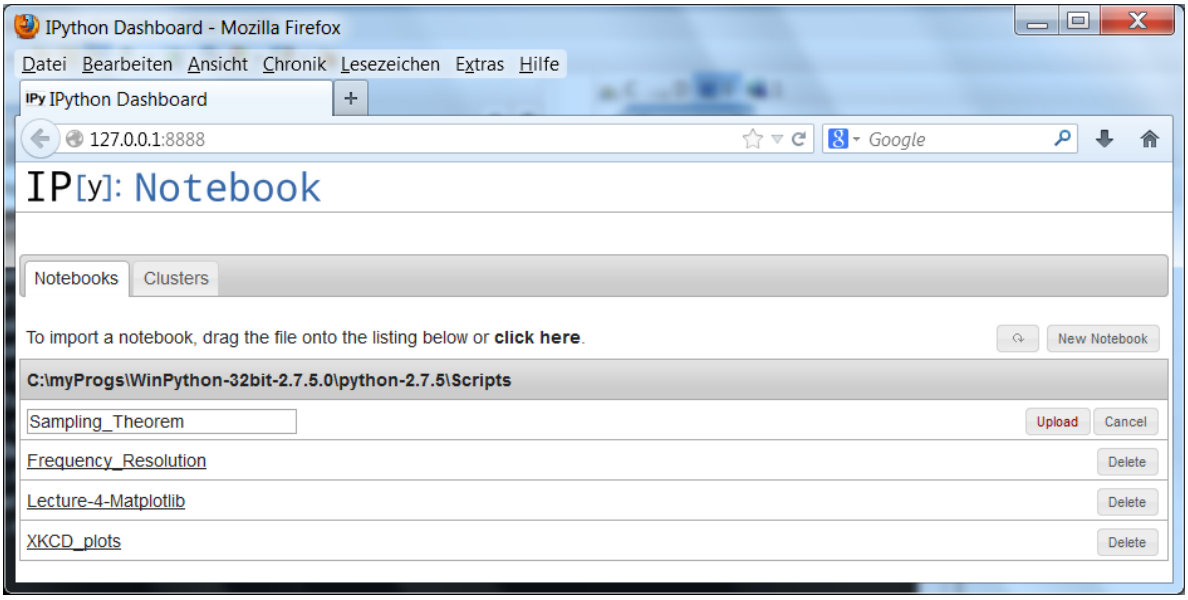


Abb. 7.2.: IPython Dashboard nach dem Start

Achtung: Wenn man versucht, mit Firefox ein Notebook im Web mit „Ziel speichern unter“ auf die eigene Festplatte zu kopieren, wird es bei manchen Links als statisches HTML abgelegt (???)! Workaround: In Git das gesamte Repository als zip-File herunterladen oder den File anklicken und den Text kopieren.

7.2.2. Beispiele für den Einsatz von IPython-Notebooks

Teaching with IPython: JupyterHub https://bids.berkeley.edu/resources/videos/teaching-ipythonjupyter-notebooks-and-jupyterhub

Brian Granger: All About Jupyter auf der NYC PyData 2015 (https://www.youtube.com/watch?v=GMKZD1Ohlzk)

Rosham Jupyter Videos Playlist Fokus auf der Bedienung von Jupyter (https://www.youtube.com/watch?v=Rc4JQWowG5I&list=PLRJx8WOUx5XcDMOxSQegCJUjTJePTlF9Z)

Seven Ways of Running IPython Notebooks Guter Überblick, https://blog.ouseful.info/2014/12/12/seven-ways-of-running-ipython-notebooks/

Powered by Jupyter: A Survey IBM Technology Blog, 2016, http://blog.ibmjstart.net/2016/03/21/powered-by-jupyter/

(I)Pythonkurse für blutige Anfänger: U.a. findet man hier die kompletten Unterlagen zum Kurs „Data Analysis and Computational Methods with Matlab™“ - der allerdings inzwischen mit IPython durchgeführt wird: https://github.com/schryer/python_course_material

Signalverarbeitung Der Blog „Python for Signal Processing (http://python-for-signal-processing.blogspot.de) wurde mit IPython Notebooks erstellt und zeigt eindrucksvoll

die hohe Darstellungsqualität von Formeln, Text und Grafiken. Und alle Notebooks können interaktiv bearbeitet werden (https://github.com/unpingco/Python-for-Signal-Processing).

Physik Grundkurs an der Uni Ulm: Die Notebooks hier sehen nicht ganz so schön aus, demonstrieren dafür aber, wie IPython auch in der Grundlagenlehre eingesetzt werden kann: http://elim.physik.uni-ulm.de/?page_id=1166

Einsatz als interaktives Laborbuch: http://www.randalolson.com/2012/05/12/a-short-demo-on-how-to-use-ipython-notebook-as-a-research-notebook/

7.2.3. Einrichten von Notebook-Servern

Für die Lehre liegt der Charme von IPython darin, es auf einem vom Intra/Internet erreichbaren Server laufen zu lassen und so die Einstiegshürden für die Studierenden zu minimieren. Das lässt sich entweder mit einem eigenen Server erreichen oder indem man auf Cloud-based Services aufsetzt:

Einrichten eines (lokalen) IPython Notebook Servers

Für den Einsatz in Classroom-Umgebungen ist

7.3. (I)Python in der Cloud

Die Installation eines eigenen IPython - Servers ist mit Sicherheitsrisiken und/oder Kosten verbunden. Aufgrund der Popularität bieten die größeren Web Service Provider aber vorgefertigte Lösungen für die Installation von (I)Python an (http://www.analyticsvidhya.com/blog/2015/09/scalable-data-science-cloud-python-r/):

Amazon Elastic Cloud Computing (EC2) (http://aws.amazon.com/de/ec2/) mit Amazon Machine Images (AMI) ermöglicht es sehr schnell einen Server mit IPython in der Cloud aufzusetzen. AMIs sind vorkonfigurierte Betriebssysteme und Software, die über Label eindeutig identifiziert werden. Für geringe Anforderungen gibt es kostenfreie Angebote: http://aws.amazon.com/de/free/.

Create and Configure a VM on Windows Azure and deploy IPython: Schritt für Schritt Anleitung vom Windows Azure SDK team (https://github.com/WindowsAzure/azure-content/blob/master/DevCenter/Python/Tutorials/azure-ipython.md). Microsoft stellt ein kostenloses e-book (http://blogs.msdn.com/b/microsoft_press/archive/2015/04/15/free-ebook-microsoft-azure-essentials-azure-machine-learning.aspx) und ein MOOC in seiner virtuellen Akademie (https://www.microsoftvirtualacademy.com/en-us/training-courses/getting-started-with-microsoft-azure-machine-learning-8425?l=ehQZFoKz_7904984382) zur Verfügung.

Wakari (https://www.wakari.io/) ist eine bequeme und kostengünstige Alternative zum Ausprobieren von Continuum Analytics (Online-Doku: https://www.wakari.io/docs/) - empfohlen! Ein oder mehrere Notebooks (bundle) können geteilt werden, siehe https://wakari.io/docs/sharing.html.

Datajoy (https://www.getdatajoy.com/) mit kostenloser Option und Angeboten für Academia.

PythonAnywhere (https://www.pythonanywhere.com/), leider noch kein IPython und Grafik. Fokus auf Websites mit Data Stack.

DominoDataLab (https://www.dominodatalab.com/) wurde entwickelt, um Data Science in der Cloud zu betreiben, IPython / Jupyter Notebooks werden unterstützt. Keine kostenlose Option.

Trinket (https://trinket.io/, http://blog.trinket.io/why-python/ und https://hourofpython.com).

Koding (https://koding.com/), stellt virtuelle Maschinen in der Cloud zur Verfügung. Die IDE ist komfortabel und hat sogar Python-Snippets. Die Infratstruktur setzt auf kostenlosen Amazon EC2 t2.micro Instanzen auf. Angeblich ist es möglich, Matplotlib über VNC zu verwenden, ich habe es nicht stabil geschafft.

7.3.1. Notebooks im Web mit Binder

Binder (http://mybinder.org/) hat den Claim: „Verwandle Dein GitHub Repo in eine Sammlung von interaktiven Notebooks“, d.h. Jupyter Notebooks auf GitHub können einfach im Webbrowser angeschaut, editiert und ausgeführt werden. Die eigentliche Rechenpower wird von Kubernetes (http://kubernetes.io/) bereit gestellt, einem Open-Source System „for automating deployment, scaling, and management of containerized applications“.

Das geht genauso mit Ruby und R.

Beispiele:

Digital Signal Processing (Sascha Spors), Masterkurs an der Uni Rostock https://app.mybinder.org:80/2741783794/notebooks/index.ipynb

7.4. Interaktive Plots mit Plotly

Mit Matplotlib gerenderte Plots sind in IPython leider statisch; man kann also nicht zoomen oder den Ausschnitt verschieben. Einen Ausweg bietet Plotly (https://plot.ly/), ein webbasierter Renderer mit APIs nicht nur für Python und IPython, sondern auch u.a. für Matlab™ oder sogar für Arduino und Raspberry Pi. Die Daten werden zu einem Plotly Server übertragen (Account benötigt), dort gerendert und als URL zur Verfügung gestellt.

Überblick und Einführung als IPython Notebook: http://nbviewer.ipython.org/github/plotly/IPython-plotly/blob/master/Plotly%20Quickstart.ipynb

7.5. Potenzielle Probleme

7.5.1. Mathjax

Wenn man zum ersten Mal versucht, ein IPython Notebook ohne Internetzugang zu starten, wird man vermutlich eine Fehlermeldung bekommen wie
„Failed to retrieve MathJax from 'http://cdn.mathjax.org/mathjax/latest/MathJax.js' Math/LaTeX rendering will be disabled."

Die benötigte MathJax Bibliothek kann aber für den Offline-Betrieb leicht vom Python / IPython Prompt aus installiert werden mit:

```
>>> from IPython.external import mathjax; mathjax.install_mathjax()
```

Da versucht wird MathJax in das IPython Source Directory zu installieren, braucht man hierfür ggf. Admin-Rechte.

Alternativ kann man den Notebook Server auch ohne Latex-Rendering starten mit

```
\$ipython notebook --no-mathjax
```

und so den Dialog bzw. die Fehlermeldung komplett unterdrücken.

7.5.2. Images im Notebook

7.5.3. Startverzeichnis von IPython

7.6. Koding

Koding (https://koding.com) ist eine Cloud-basierte Entwicklungsplattform, mit der man sofort loslegen kann zu programmieren ohne auf dem eigenen Computer etwas zu installieren.

Koding erlaubt Real-Time Collaboration, alle Dokumente etc. können geteilt werden, es gibt außerdem einen Video-Chat.

In Kodings virtueller Maschine (grünes Kästchen in der linken Sidebar) läuft Ubuntu Linux (14.2, Dezember 2015), vorinstalliert sind u.a. Python, Ruby, perl, gcc, ..., weitere Software kann nachinstalliert werden mit den üblichen Tools oder mit Kodings Paketmanager (kpm). Die VM bringt 1 GB RAM und einen Core mit sowie 3 GB Festplattenspeicher zur Verfügung, die schnell knapp werden (siehe Optionsfeld links neben dem Namen der VM).

7.6.1. Start

Ein erstes „Hello World“-Programm in verschiedenen Sprachen: http://learn.koding.com/guides/hello-world/

3. If you want to install more software, take a look at Koding Package Manager (kpm). It can make life much easier when it comes to installing new stuff. http://learn.koding.com/guides/getting-started-kpm/

4. To run a command as the ‘root‘ user, prefix any command with ‘sudo <command>‘. But remember, with great power, comes great responsibility! :-)

5. By default, your sudo password is blank. Most people like it that way but if you prefer, you can use the ‘sudo passwd‘ command and change the default (blank) password to something more secure. This is completely optional.

7.6.2. Einrichtung von Python

Startet man Kodings virtuelle Maschine das erste Mal, sind Python 2 und 3 bereits installiert. Python 2 startet man vom Terminal mit `python`, Python 3 mit `python3`.

`sudo apt-get install libace-perl` installiert den einfachen Editor ACE

Weitere Module installiert man mit dem Koding Paket Manager `kpm`, z.B. den Python Installer `pip`:

```
kpm install pip
```

Entfernen kann man Pakete mit `kpm uninstall xxx`.

Ubuntus Paketmanager `apt` kann ebenso verwendet werden.

```
sudo apt-get install python-dev
sudo pip install numpy (reicht zum Kaffeekochen)
sudo apt-get install libfreetype6-dev libxft-dev (+ 10 MB)
sudo apt-get install tk tk-dev
sudo apt-get install ipython ipython-notebook
sudo pip install matplotlib
```

7.6.3. VNC

Installation und Setup von VNC mit dem Koding Package Manager (kpm)
http://learn.koding.com/guides/vnc-startup-guide/
https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-vnc-on-ubuntu-14-04

a) Downloade, installiere und starte TightVNC Server mit XFCE4 Desktop:

`kpm install vnc`

Achtung: Dieser Schritt reicht bereits, um den VNC-Server zu installieren und zu starten, das Tutorial ist hier etwas missverständlich. Während der Installation muss ein Password eingegeben werden - unbedingt aufschreiben, ohne PW kann man sich später nicht mit dem VNC Server verbinden! Der Server wird automatisch nach Ende der Installation gestartet, manuell würde das mit `vncserver :1` geschehen.

Man kann den Server stoppen mit `vncserver -kill :1`.

b) Downloade, installiere, and starte WebSockify: `kpm install novnc`

WebSockify ist ein VNC Client, der HTML5 verwendet. Damit wird WebSockets Support für beliebige Anwendungen/Server bereitgestellt: Es übersetzt WebSockets Traffic zu normalem Socket Traffic.

WebSockify wird gestartet mit `novnc`.

Anpassungen in `/home/YOUR_USER_NAME/.vnc/xstartup`

c) Verbinde mit dem VNC - Server: `http://YOUR_ASSIGNED_URL:8787/vnc.html?host=YOUR_ASSIGNED_URL&port=8787`

7.6.4. Rama dama!

Eine Liste der installierten Pakete, geordnet nach Größe erhält man mit:
`dpkg-query -W --showformat='${Installed-Size} ${Package}\n' | sort -nr | less`

Wieviel Platz noch frei ist, erfährt man mit `df -h`, die 10 größten Subdirectories im aktuellen Directory werden gelistet mit:

`du -sk * | sort -nr | head -10`

Große Files findet man mit

`find / -type f -size +1024k` oder

`find / -size +50000  -exec ls -lahg {} \;`

Aufgeräumt wird mit

`sudo apt-get clean` (löscht heruntergeladene *.deb Pakete, die bereits installiert wurden)

`sudo apt-get autoclean` (löscht alte Paketversionen)

`sudo apt-get autoremove` (löscht verwaiste Files nach Installationen)

8. Am Anfang war der Vektor: Listen, Tuples, Arrays ...

Das Besondere an Sprachen für Computational Mathematics wie Matlab^{TM}, Scilab, R und Octave ist deren Fähigkeit, umfangreiche numerische Daten, Vektoren, Matrizen oder Tensoren effizient zu repräsentieren und zu manipulieren. Das ermöglicht schnelles „number crunching“ für Problemstellungen der linearen Algebra. Python hat dies von Haus aus nicht, diese Funktionalität wird durch das Modul **NumPy** und dessen Datentyp *N*-dimensional array (ndarray) nachgerüstet. Die NumPy Bibliothek stellt zahlreiche Funktionen wie `sqrt()` und `sin()` für sehr schnelle ein- und mehrdimensionale Arrayberechnungen bereit.

An manchen (wenigen) Stellen merkt man, dass die Arraymathematik nachträglich angeflanscht wurde, dort bietet Matlab^{TM} z.T. kompaktere Schreibweisen mancher Operationen. Für den Anfänger ist es manchmal verwirrend, dass es Funktionen mit ähnlichen oder sogar gleichem Namen in der „Python-Grundausstattung“ und im numpy-Modul gibt. Die Funktion `max(a,b)` gibt z.B. einen Skalar zurück, während die Funktion `numpy.maximum(a,b)` auch Vektoren a und b akzeptiert und einen Vektor mit den elementweisen Maxima zurückliefert.

Detaillierte Unterschiede kann man unter http://wiki.scipy.org/NumPy_for_Matlab_Users nachlesen.

8.1. Iterierbare Datentypen

8.1.1. Array / NumPy Array

Arrays sind Listen mit einer *festen Größe*, die aus *numerischen Elementen vom gleichen Typ (= homogen)* wie Characters, Integers oder Floating Point Zahlen bestehen und daher effizient im Speicher abgelegt werden können. Die Größe von Arrays darf daher nach der Definition / Erzeugung nicht mehr geändert werden[1] Das Standard Modul `array` stellt zwar auch einen Datentyp Array zur Verfügung, für numerische Anwendungen sollte aber unbedingt das Numpy Modul und dessen `ndarray` bevorzugt werden aufgrund der viel leistungsfähigeren Arithmetik.

Die Operationen `+, -, *, /` werden elementweise zwischen Arrays ausgeführt, die daher gleiche Dimensionen haben müssen, in Matlab^{TM} entspricht das den „punktierten“ Operationen `.+` , `.-` , `.*`, `./`. Für das Skalarprodukt benutzt man den Befehl `np.dot(a,b)`. Viele weitere elementweise Array-Operationen sind im numpy und scipy - Modul definiert wie `sin(a), exp(a)` etc.

Eine sehr gute Einführung in numpy Arrays gibt http://wiki.scipy.org/Tentative_NumPy_Tutorial.

[1] Bei Matlab^{TM} darf die Größe eines Arrays dynamisch geändert werden; solche Konstrukte führen dann zu Problemen beim Portieren zu Python. Sie sollten aber auch in Matlab^{TM} vermieden werden, da sie zu massiven Performanceeinbußen führen.

Erzeugung

Sehr häufig werden Arrays mit gleichmäßig abgestuften Elementen benötigt:

`a = np.arange(start, end, step)`
erzeugt linear abgestufte Arrays mit vorgegebenem Inkrement, z.B.
`a = np.arange(0, 2, 0.1) = [0, 0.1, 0.2, ..., 1.8, 1.9]` - ohne Endpunkt (hier: 2)! Die Argumente `start` (Default: 0) und `step` (Default: 1) sind optional. Numerische Ungenauigkeiten können bei nicht-ganzzahligem Inkrement (`step`) dazu führen, dass das letzte Element des Arrays größer als `end` ist. Hier sollte man besser `linspace` verwenden.

`a = np.linspace(start, end, num)`
erzeugt linear abgestufte Arrays mit vorgegebener Anzahl, z.B.
`a = np.linspace(0, 2, 21) = [0, 0.1, 0.2, ..., 1.9, 2.0]` - mit Endpunkt (hier: 2)! Durch `endpoint = False` kann der Endpunkt ausgeschlossen werden.

`a = np.logspace(start, end, num)`
erzeugt logarithmisch abgestufte Werte, z.B.
`a = np.logspace(-1, 2, 4) = [0.1, 1.0, 10.0, 100.0]`, die Basis des Exponenten (Default: `base=10`) und `endpoint=False` können dabei optional gewählt werden.

Platz für Arrays kann z.B. auf die folgenden Arten reserviert werden:

```
a = np.empty(1024) # Reserviere Platz für 1024 Float - Elemente
a = np.ones((3,4), dtype = int) # 3 x 4 Matrix mit 1 gefüllt im Integerformat
a = np.zeros(10, dtype = 'c8') #  Array mit Nullen gefüllt, im Format complex 64 Bits
a = randn(5,5) # 5 x 5 Array mit normalverteilten Floats (Varianz = 1)
```

Schließlich können auch geeignete (d.h. mit Elementen von gleichem Typ) Listen / Tuples / Strings in numpy Arrays umgewandelt werden:
`numpy.asarray(my_list)`

8.1.2. Vervielfältigen von Arrays

`np.tile(a,(2,2))` „kachelt“ ein eindimensionales Array `a` zweimal in die x- und zweimal in die y-Richtung. `a` kann auch mehrdimensional sein und es kann auch in mehr Dimensionen gekachelt werden.

`np.repeat(a, N)` klopft das Array `a` zunächst flach und wiederholt dann die einzelnen Elemente `N` mal. Bei Arrays geht das im Gegensatz zu Listen auch in mehreren Dimensionen.

8.1.3. Array-Datentypen

Eine Auswahl der möglichen Datentypen ('dtypes') für ndarrays ist in Tab. 8.1 dargestellt, z.B. 'f' für einen Vier-Byte Float (Default) und 'i' für einen vorzeichenbehafteten Integer mit zwei Bytes.

8.1.4. Weitere iterierbare Datentypen

Bei Python gibt es außerdem die Datentypen Tuple, Liste und String, die sich ähnlich, aber nicht gleich wie Arrays verhalten. Das ist zwar jedem Programmierer klar, aber in Matlab™ gibt es diese Unterschiede nicht.

Die Gemeinsamkeit aller folgenden Datenstrukturen ist, dass sie „Container“ für eine geordnete Folge von Elementen sind. Die einzelnen Elemente lassen sich über ihren Index ansprechen, sie sind „iterierbar“. Bei seltsamen Fehlermeldungen sollte man zuerst sicherstellen, ob die Variablen wirklich den Typ und die Form haben, die man gerne hätte. Hierfür sind die folgenden Befehle nützlich:

```
x = np.ones((2,4))                  # Numpy Array der Form 2 x 4, gefüllt mit '1'
type(x) >>> <type 'numpy.ndarray'>
x.size = size(x)   >>> 8            # Gesamtanzahl Elemente
x.shape = shape(x) >>> (2,4)
len(x) >>> 2                        # Vorsicht: Das ist nur die Länge der ersten ACHSE
```

Schließlich gibt es noch „Dictionaries“, die nicht über die Reihenfolge, sondern über „Keys“ indiziert werden.

8.2. Indizierung und Slicing

Indizierung nennt man es, wenn auf *einzelne* Elemente eines Arrays (bzw. Tuples, Strings, Dictionaries oder einer Liste) zugegriffen wird, also z.B. `a = x[23]`. Das erste Element wird mit `a = x[0]` angesprochen, das letzte mit `a = x[len(a)-1]` oder kurz `a = x[-1]`.

Slicing nennt man den Vorgang, auf einen *Teilbereich* einer Liste, eines Tuples, Strings oder Arrays zuzugreifen. Die generelle Syntax dafür ist `a = x[lo:hi:step]` (eckige Klammern!). `lo` ist der Index für das erste Element aus `x`, das nach `a` kopiert wird, der `hi` Wert kennzeichnet das erste Element, das *nicht* nach `a` übertragen wird. Die Differenz `hi - lo` entspricht daher der Anzahl der übertragenen Werte (für den Default `step` = 1). Der Parameter `step` gibt die Schrittweite vor, mit der die Elemente von `a` ausgewählt werden.

Die Syntax ist für Tuples, Listen, Strings und Arrays identisch, alle Angaben sind optional:

Indizes zeigen auf die Elemente selbst, bei Slices zeigt die untere Grenze auf die linke Trennlinie und die obere Grenze auf die rechte Trennlinie zwischen den Elementen, zurückgegeben werden die Elemente dazwischen, optional mit Schrittweite > 1 oder -1:

```
|                                          Index             Slice
Index von hinten:  -6  -5  -4  -3  -2  -1  a = [a,b,c,d,e,f]  a[1:] == [b,c,d,e,f]
Index von vorn:     0   1   2   3   4   5  len(a) == 6       a[:5] == [a,b,c,d,e]
|                +- -+- -+- -+- -+- -+- -+  a[0] == a         a[:-2] == [a,b,c,d]
|                | a | b | c | d | e | f |  a[5] == f         a[1:2] == [b]
|                +- -+- -+- -+- -+- -+- -+  a[-1] == f        a[1:-1] == [b,c,d,e]
Slice von vorn:   :   1   2   3   4   5   :  a[-2] == e        a[:] == [a,b,c,d,e,f]
Slice von hinten: :  -5  -4  -3  -2  -1   :                   a[1:5:2] == [b,d]
```

nach: http://wiki.python.org/moin/MovingToPythonFromOtherLanguages

Python Indizes und Slices für eine Liste mit sechs Elementen.

```
import numpy as np
x = (2, 3, 5, 7)      # Tuple
x = range(9)          # Liste [0, 1, ... , 8]
x = np.arange(9)      # = arange(0,9,1); Array [0, 1, 2, ..., 8] - ohne letztes Element!
#
a = x[lo:hi]          # Elemente von lo bis hi-1
a = x[lo:]            # Elemente ab lo bis zum Ende des Arrays
a = x[:hi]            # Elemente vom Anfang bis hi-1
a = x[:1]             # nur erstes Element, gleiches Ergebnis wie x[0]
a = x[:]              # Eine Kopie des gesamten Arrays
x[::2] = np.zeros(np.ceil(len(x)/2.)) # Setze jedes zweite Element = 0
```

Negative Werte für `lo` oder `hi` beziehen sich auf den *Endpunkt* anstatt auf den *Anfang* des Arrays `x`. Negative Werte für den `step` Paramenter lassen die Slicing-Operation rückwärts zählen:

```
a = x[-1]    # letztes Element im Array
a = x[-2:]   # letzte 2 Elemente im Array (= von Ende-2 bis zum Ende)
a = x[:-2]   # alles außer der letzten zwei Elemente
a = x[::-1]  # dreht die Reihenfolge um (step = -1)!
             # Alternativ (nur für Listen): a = x.reversed()
a = x[::-2]  # >>> [8, 6, 4, 2]
a = x[1:-1:-2]  # >>> [] # bei negativer Schrittweite hi und lo vertauschen!
a = x[-1:1:-2]  # >>> [8, 6, 4, 2]
```

8.2.1. Aufgemerkt!

Python ist (zu?) nett zum Programmierer: Wenn das ursprüngliche Array weniger Elemente hat als für die Slicing-Operation benötigt, wird eine leere Liste ohne Fehlermeldung zurückgegeben. Ist z.B. `a = "'b"'`, dann ergibt `a[2] == []`. Ist dieses Verhalten unerwünscht, muss eine zusätzliche Abfrage implementiert werden.

Bemerkungen:

- Slicing bewirkt in Python ein „shallow copy“, d.h. die Operationen `b = a[:]` oder `b = a[3:9]` referenzieren nur auf Elemente von `a`, erzeugen aber keine Kopie von `a` im Speicher - ändert man `b`, ändern sich auch die entsprechenden Elemente von `a`. Matlab^{TM} verhält sich anders, hier wird immer eine Kopie erzeugt!
- Man kann in einem Schritt Slicen und Indizieren: `x = range(100); a = x[::10][5]` fasst jedes zehnte Element von `x` in einer neuen Liste zusammen und adressiert davon das fünfte - in diesem Fall der Wert 50.

Ausführliche Diskussion zu diesen Themen: http://stackoverflow.com/questions/509211/good-primer-for-python-slice-notation

Mehrdimensionale Arrays

Mehrdimensionale Arrays in Numpy kann man ebenso leicht erzeugen, u.a. durch Angabe eines entsprechenden Tuples in `ones, zeros, empty ...`, siehe auch (siehe auch http://pages.physics.cornell.edu/~myers/teaching/ComputationalMethods/python/arrays.html):

```
# Erzeuge 10 x 10 - Array mit normalverteilten Zufallszahlen, m = 1, sigma = 0.2
xx = np.zeros((10,10))
xx = np.random.normal(loc=1.0, scale=0.2, size=(10,10))
aa = xx[1, 2]           # Auswahl eines Elements
aa = xx[1:3, 4:6]       # Auswahl eines Teilbereichs
aa = xx[:,1]      # Auswahl der zweiten Spalte
aa = xx[-1,:]                   # Auswahl der letzten Zeile
aa = xx[::2, ::2]       # Dezimation um 2 in beiden Achsen
```

Mit Hilfe von `meshgrid` kann man in Python und Matlab^{TM} Funktionen von zwei Variablen vektorisieren. `meshgrid(x,y)` erzeugt aus den eindimensionalen Arrays `x` und `y` zweidimensionale Arrays `XX` und `YY` mit den Dimensionen `len(x)` x `len(y)`. Dabei wird der erste Vektor zeilenweise wiederholt und der zweite spaltenweise. Die Variable ZZ im obigen Beispiel enthält alle Kombinationen von x and y. Für DSP - Anwendungen kann man so sehr schön komplexe Koordinaten generieren. [XX muss je nach Anwendung noch transponiert werden, XX = XX.T ???]

Im Gegensatz zu `meshgrid` unterstützt die Funktion `mgrid` auch mehr als zwei Dimensionen. Die Syntax ist allerdings ziemlich kryptisch: Man kann die Schrittweite vorgeben (wie bei `arange`); die Schrittzahl (wie bei `linspace`) gibt man durch die imaginäre Einheit vor ...

```
x = np.asarray([1, 2, 3])  # oder
x = np.linspace(1, 3, 3) # oder
x = np.arange(1, 4) # ohne letztes Element!
y = np.asarray([1, 2])

XX, YY = np.meshgrid(x, y)
>>> XX
array([[1, 2, 3],
       [1, 2, 3]])
>>> YY
array([[ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])

ZZ = XX.T + YY.T*1j
>>> ZZ =>
```

```
array([[ 1.+1.j,  1.+2.j],
       [ 2.+1.j,  2.+2.j],
       [ 3.+1.j,  3.+2.j]])

# Alternative mit gleichem Resultat:
#   spezifiziere SchrittWEITE (vgl. arange)
XX, YY = np.mgrid[1:4:1, 1:3:1]
# oder SchrittZAHL (vgl. linspace)
XX, YY = np.mgrid[1:3:3j, 1:2:2j] # step = komplex
ZZ = XX + YY * 1j
```

Lst. 8.1: Multidimensionale Arrays in Python

8.2.2. Listen

Listen sind Datenstrukturen, die aus einer Folge von einem oder mehr Elementen bestehen. Die Elemente können Zahlen oder wiederum Listen (wie z.B. Strings) oder eine Mischung daraus sein. Listen werden angeben durch **eckige Klammern**, die einzelnen Elemente werden durch Kommas separiert. Numerische Listen lassen sich außerdem bequem mit dem Befehl `range()` erzeugen:

Beispiele:
`a = range(9); b = [1, 'a', 2.e-3, [1.5, 2]]`

`b = [3]` oder `b = [3,]` erzeugen eine Liste mit einem Element '3'.

Listen sind iterierbare, veränderliche („mutable“) Objekte und können über die folgenden Methoden (Ausgabe über `dir([])`) *in-place* modifiziert werden, d.h. ohne zusätzlichen Speicher zu belegen. Das funktioniert *nicht* mit Strings und Tuples, da diese Objekte nicht veränderlich sind. Im folgenden sollen `a, L` Listen und `x` ein Listenelement sein, `i, j, k` sind vom Typ Integer.

a[i] = x Das Element mit dem Index `[i]` wird durch `x` ersetzt.

a[i:j] = L Die Slice von `i` bis `k` wird durch die Liste `L` ersetzt.

a[i:j:k] = L Ersetze die Elemente `a[i:j:k]` durch die Liste `L` (muss die gleiche Länge haben wie Liste der zu ersetzenden Elemente).

Die folgenden Operationen ändern die Größe einer Liste und sind daher nicht für Arrays definiert.

del a[i:j] Lösche die Slice von `i` bis `k`, äquivalent zu `a[i:j] = []`.

del a[i:j:k] Lösche die Elemente `a[i:j:k]` aus der Liste.

a.append(x) Hänge ein Element ans Ende der Liste an; äquivalent zu `a[len(a):len(a)] = [x]`.

a.extend(L) Erweitere die Liste `a` durch Anhängen aller Elemente der Liste `L`; äquivalent zu `a[len(a):] = L`.

a.insert(i, x) Füge ein Element `x` vor der Position `i` ein. `a.insert(0, x)` fügt das Element am Anfang der Liste, `a.insert(len(a), x)` am Ende der Liste ein (äquivalent zu `a.append(x)`). Gleichbedeutend mit `a[i:i] = [x]`.

a.remove(x) Entferne das erste Element der Liste mit dem Wert `x`. Existiert kein Element `x` in der Liste, wird ein `ValueError` erzeugt.

pop([i]) Entferne das Element an Position `i` gib es zurück. Wenn kein Index angegeben wird, entfernt `a.pop()` das letzte Element der Liste und gibt es zurück. Die eckigen Klammern bedeuten hier, dass `i` optional ist.

a.index(x) Gibt den Index des ersten Elements mit Wert `x` zurück. Existiert kein Element `x` in der Liste, wird ein `ValueError` erzeugt.

a.count(x) Gibt zurück, wie oft `x` in der Liste enthalten ist.

a.reverse() Drehe die Reihenfolge der Listenelemente um.

a.sort() Sortiere die Listenelemente.

a.sort([cmp[, key[, reverse]]]) Sortiere die Listenelemente mit zusätzlichen Optionen (eckige Klammern). Mit `cmp` kann eine eigene Suchfunktion mit zwei Eingangsargumenten angegeben werden, die 0, +1 oder -1 zurückgibt. `key` gibt eine Funktion an, die auf jedes Element vor der Sortierung angewendet wird. `key = str.lower` wandelt z.B. alle Elemente in Kleinbuchstaben um, `key = lambda x: x[2]` definiert eine Inlinefunktion („lambda“), die Spalte 2 der Elemente zurückgibt, so dass nach dieser Spalte sortiert wird. Mit `reverse = True` wird in absteigender Reihenfolge sortiert. Siehe auch http://wiki.python.org/moin/HowTo/Sorting/.

Mit Listen sind keine arithmetischen Operationen möglich; `a = 3 * L` ergibt eine Liste, die aus drei Kopien von `L` besteht, `L = a + b` verknüpft die beiden Listen `a` und `b` miteinander. Für Vektor / Matrix-Arithmetik muss man mit Arrays arbeiten (s.u.). Bei Bedarf kann man mit `a = np.asarray(L)` eine Liste in ein Array umwandeln.

8.2.3. Tuples

Tuples sind „Listen für Arme“, sie werden angelegt mit `a = (1,2,'a')` (runde Klammern!). Sie haben nur wenige Methoden - `dir( ('a','b') )` ergibt nur `'count'` und `'index'` und sie sind unveränderbar („immutable“). a[2] = 'x' ergibt daher einen Fehler. Versuchen Sie selbst, was das Kommando `def()` für eine allgemeine Liste ergibt. Der Vorteil von Tuples gegenüber Listen ist aber genau, dass sie so wenig können: Sie sind einfacher für den Interpreter zu verwalten und benötigen daher weniger Rechenaufwand. Sie werden Python-intern immer benutzt, wenn mehrere Objekte in einen Container gestopft werden sollen, z.B. bei Funktionen, die mehrere Werte zurückgeben. Besonderheit: Ein Tuple mit einem Element muss mit `a = (1,)` erzeugt werden - `a = (1)` ist das gleiche wie `a = 1`.

Tuples können umgewandelt werden in Listen mit `a = list(my_tuple)`.

8.2.4. Strings

Strings sind Listen von Zeichen und Escape-Sequenzen (z.B. `\t`, `\n` und `\f` für Tabulator, Line Feed bzw. Seitenvorschub, `\xCA` für einen Hex-Wert und `\077` für einen Oktalwert), sie werden erzeugt, indem man eine Folge von Zeichen mit einem (gleichen!) Paar von einzelnen oder doppelten Anführungszeichen einschließt (kein Unterschied). Dabei darf innerhalb des

Strings jeweils die anderen Variante als Zeichen verwendet werden, z.B. `"Susi's Sushi"` oder `'"HoHoHo" sagte der Osterhase'`.

Strings sind genau wie Tuples unveränderbar, haben aber einige spezielle Methoden zur Textbearbeitung, z.B. wandelt `b = a.upper()` alle Buchstaben von a in Großbuchstaben um. Sollen die Sonderzeichen im string nicht ausgewertet werden, muss der String als „raw" deklariert werden: `xlab = r"$\tau_g$"` wird z.B. als TeX-String von matplotlib als τ_g dargestellt. Ohne das vorangestellte „r" wird `\t` als Tabulator interpretiert.

8.2.5. Dictionaries

Um in einem Dictionary den (ersten) Schlüssel zu einem Eintrag zu finden, kann sich mit einer List Comprehension behelfen:

```
1 key = [key for key,value in mydict.items() if value=='value' ][0]
```

Type Code	Short	Typ	Bytes
bool	'b'	Boolean (True oder False), als ein Byte gespeichert	1
int		Platform Integer (normalerweise int32 oder int64)	
int8	'i1'	Byte (-128 bis 127)	1
int16	'i2'	Integer (-32768 bis 32767)	2
int32	'i4'	Integer (-2^{31} bis $2^{31}-1$)	4
int64	'i8'	Integer (-2^{63} bis $2^{63}-1$)	8
uint8	'u1'	Unsigned Integer (0 bis 255)	1
uint16	'u2'	Unsigned Integer (0 bis 65535)	2
uint32	'u4'	Unsigned Integer (0 bis $2^{32}-1$)	4
uint64	'u8'	Unsigned Integer (0 bis $2^{64}-1$)	8
float	'f'	Kurzform für float64	8
float16	'f2'	Half precision Float: Sign Bit, 5 bits Exponent, 10 bits Mantisse	2
float32	'f4'	Single precision Float: Sign Bit, 8 bits Exponent, 23 bits Mantisse	4
float64	'f8' oder 'd'	Double precision Float: Sign Bit, 11 bits Exponent, 52 bits Mantisse	8
complex	'c'	Kurzform für complex128	16
complex64	'c8', 'g'	Komplexe Zahl, repräsentiert durch zwei 32-bit Floats	8
complex128	'c16', 'G'	Komplexe Zahl, repräsentiert durch zwei 64-bit Floats	16

Tab. 8.1.: Datentypen für Numpy Skalare und Arrays

9. Interessante Module und Projekte

9.1. Sympy - symbolische Algebra

Das Python Modul für symbolische Algebra (http://sympy.org/) ist viel zu umfangreich, um hier im Detail darauf eingehen zu können. Einen ersten Eindruck können Sie mit dem IPython Notebook http://nbviewer.ipython.org/urls/raw.github.com/ipython/ipython/master/examples/notebooks/SymPy%20Examples.ipynb bekommen.

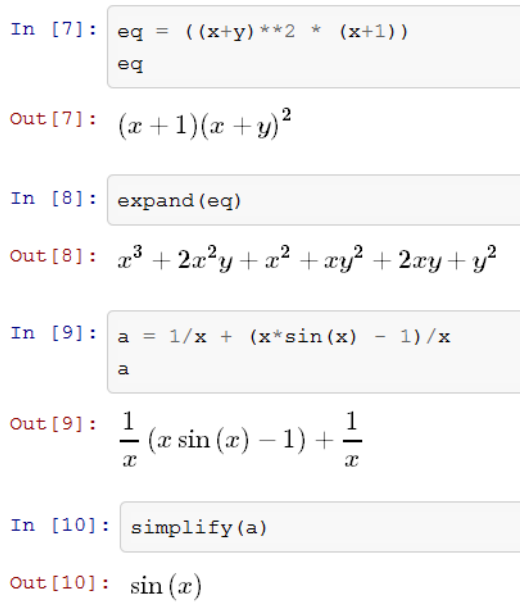


Abb. 9.1.: Ausschnitt aus einer Sympy Session in IPython

Ein tolles Angebot sind auch **Mathics** (http://www.mathics.net/) und **Sympy Live** (http://live.sympy.org/); hier können Sie online und live mit Sympy arbeiten!

Weitere Tutorials:
http://nbviewer.ipython.org/urls/raw.github.com/jrjohansson/scientific-python-lectures/master/Lecture-5-Sympy.ipynb

http://mattpap.github.io/scipy-2011-tutorial/html/index.html

Aufbauend auf SymPy wurde Mathics (http://www.mathics.org/) entwickelt, mit Mathematica - kompatibler Syntax und Funktionen. Mathics kann auch online verwendet werden (Abb. 9.2) und erlaubt sogar 2D- und 3D-Plots!

Achtung: Sympy kann man leicht verwechseln mit ...

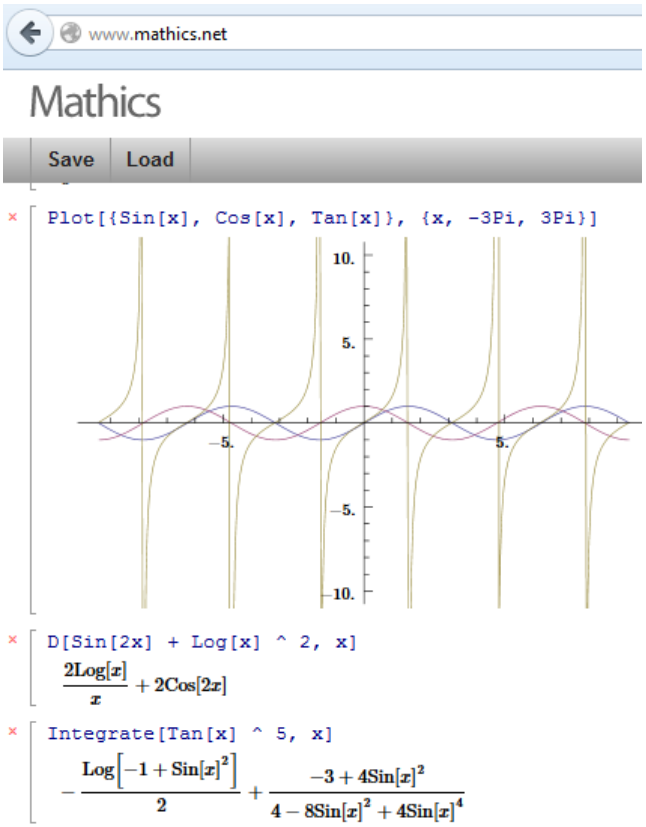


Abb. 9.2.: Ausschnitt aus einer Mathics Online-Session

9.2. Simpy - eventbasierte Simulationen

Simpy (https://simpy.readthedocs.org/) ist ein prozessbasiertes Framework zur Simulation von Systemen mit diskreten Ereignissen (*Discrete Event Systems*).

DEVSimPy (http://code.google.com/p/devsimpy/) ist ein GUI zur Modellierung und Simulation von Systemen. Simpy setzt auf der *Discrete EVent System Specification* (DEVS, http://en.wikipedia.org/wiki/DEVS) auf.

In http://de.scribd.com/doc/51553388/DEVS-for-continuous-system-simulation-application-to-asynchonous-machine wird z.B. eine Asynchronmaschine als Quantized State-System (QSS) modelliert und simuliert, die Studienarbeit nutzt DEVSimpy explizit als Alternative zu Simulink.

9.3. Audio

Es gibt zahlreiche Libraries, Wrapper und Applikationen für Audio mit Python - siehe https://wiki.python.org/moin/PythonInMusic. Hier sind nur die gängigsten Libraries und Anwendungen für Musikwiedergabe und -aufnahme aufgelistet.

Unter http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/Samples.html gibt es WAV-Files in verschiedensten Formaten, mit denen man die Libraries testen kann.

High-End Audio in verschiedenen Formaten gibt es u.a. unter: http://www.24bit96.com/24bit96khz-download-sites/hd-music-download.html
http://www.lessloss.com/high-resolution-audiophile-recordings-c-68.html

9.3.1. Standardlibraries

Von Haus bringt Python die folgenden Audio-Libraries mit:

Mit dem **winsound** (http://docs.python.org/3/library/winsound.html) Modul kann man verschiedene Sounds und WAV-Files unter Windows abspielen:

```
1 import winsound as wsnd
2 wsnd.Beep(440,1000) # Spiele den Kammerton a (440 Hz) für 1000 ms
3 wsnd.MessageBeep(type=MB_OK) # Spiele den Windowssound "'OK"' ab (oder andere Windows-
      Sounds)
4 wsnd.PlaySound("C:\Windows\Media\chord.wav",wsnd.SND_ASYNC) # WAV-File abspielen und
      sofort zurückkehren (non-blocking, ("'ASYNC"')
5 wsnd.PlaySound("SystemExit", winsound.SND_ALIAS) # Spiele den unter "'SystemExit"'
      registrierten Soundschnipsel
```

Lst. 9.1: Audioausgabe mit WinSound

Das **wave** (http://docs.python.org/3/library/wave.html) Module stellt ein Interface zum Lesen und Schreiben von WAV Soundformaten in Mono und Stereo zur Verfügung. (De)Kompression wird nicht unterstützt (`comptype` ist immer `None`). Es gibt deutlich leistungsfähigere und komfortablere Module, aber `wave` wird mitgeliefert und benötigt keine Binärlibraries. Die Dokumentation ist vollkommen unzureichend, das Beispiel unter http://mvclogic.com/python/wave-python-module-usage/ demonstriert aber die grundlegenden Funktionen:

```
1  import wave
2  # convert audio file to 8 kHz mono (mit Aliasing)
3  f = wave.open(r'C:\Windows\Media\tada.wav') # open in read mode
4  nFrames = f.getnframes() # get number of frames
5  wdata = f.readframes(nFrames) # read entire wav file
6  params = f.getparams() # tuple with (channels, ,f_S, )
7  # tuple (nchannels, sampwidth, framerate, nframes, comptype, compname)
8  f.close()
9
10 n = list(params); print n
11 n[0] = 1 # MONO
12 n[2] = 8000 # 8 kHz
13 params = tuple(n); print(params)
14
15 f = wave.open(r'D:\Daten\tada_mono.wav','wb') # open in write mode
16 f.setparams(params)
17 f.writeframes(wdata)
18 f.close()
```

Lst. 9.2: Audio-I/O mit Wave

Das **scipy.io.wavfile** Modul (http://docs.scipy.org/doc/scipy/reference/io.html) enthält die beiden Funktionen `read` und `write`, die genau das machen - WAV-Files in Numpy-Arrays zu schreiben oder zu lesen (Listing 9.3). Das funktioniert allerdings nur für 16 Bit und 32 Bit Integer-Formate!

```
1  # http://stackoverflow.com/questions/18644166/how-to-manipulate-wav-file-data-in-python
2  import numpy as np
3  from scipy.io import wavfile
4
5  rate, data = wavfile.read('M1F1-int16-AFsp.wav') # returns np array with 16b or 32b
       integers
6
7  # Take the sine of each element in `data`.
8  sindata = np.sin(data) # returns float array
9
10 # Scale up the values to 16 bit integer range and round the value.
11 scaled = np.round(32767*sindata) # still a float array
12
13 # Cast `scaled` to an array with a 16 bit signed integer data type.
14 newdata = scaled.astype(np.int16)
15
16 # Write the data to 'newname.wav'
17 wavfile.write('newname.wav', rate, newdata)
```

Lst. 9.3: WAV-File I/O mit scipy.io.wavfile

Angeblich sind diese Beschränkungen inzwischen aufgehoben (Listing 9.4, ansonsten gibt es eine Alternative unter https://gist.github.com/WarrenWeckesser/7461781 (basierend auf dem `wave`-Modul). .

```
rate, in_data = wavfile.read(in_name)
in_nbits = in_data.dtype
out_nbits = in_nbits

out_data = out_data.astype(out_nbits)
wavfile.write(out_name, rate, out_data)
```

Lst. 9.4: WAV-File I/O mit scipy.io.wavfile und beliebiger Wortbreite

Das audioop Modul enthält Funktionen wie `avg, rms, reverse` für Soundschnipsel, die mit 8, 16 oder 32 bit Signed Integer Samples in Python Strings abgelegt sind (gleiches Format wie in `al` und `sunaudiodev` Modulen verwendet). Ebenfalls enthalten sind Funktionen für Ähnlichkeitsanalyse und Resampling.

9.3.2. PortAudio und Python-Wrapper dafür

PortAudio (http://www.portaudio.com/ ist eine Cross-Plattform Audio I/O Library zum Aufnehmen und Abspielen von Audiostreams. Bei vielen Wrapper-Paketen wird sie mitinstalliert), daher benötigt man meist passende Binaries.

PyAudio

Mit **PyAudio** (http://people.csail.mit.edu/hubert/pyaudio/) kann man unter Python auf verschiedenen Plattformen leicht Audiofiles abspielen und aufnehmen (blocking / non-blocking). PyAudio ist mehr oder weniger ein Wrapper für Portaudio.

Zur Installation verwendet man am Einfachsten ein passendes (!) Binary, für Windows z.B. von Christoph Gohlke für verschiedene Python 2.x und 3.x Versionen.

Erweitertes Beispiel aus http://people.csail.mit.edu/hubert/pyaudio/docs/:

```
# Ende der gemeinsamen Import-Anweisungen
import pyaudio
import wave
wf = wave.open(r'D:\Musik\wav\Jazz\07 - Duet.wav')

np_type = np.int16 # WAV = 16 bit signed per sample, 1 frame = R + L sample

p = pyaudio.PyAudio() # instantiate PyAudio + setup PortAudio system
# open a stream on the first audio device with the parameters of the WAV-File
# for reading or writing
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True)
CHUNK = 1024 # read chunks of 1024 frames
# initialize numpy arrays
samples_in = zeros(CHUNK*2, dtype=np_type)
samples_out = zeros(CHUNK*2, dtype=np_type)
samples_l = zeros(CHUNK, dtype=np_type)
samples_r = zeros(CHUNK, dtype=np_type)
data = wf.readframes(CHUNK) # read the first CHUNK of frames

while data is not None: # read frames until end of file is reached
```

```
24      stream.write(data) # play audio by writing audio data to the stream (blocking)
25      # read frames into string, then convert to numpy array:
26      samples_in = np.fromstring(wf.readframes(CHUNK), dtype=np_type)
27      samples_l = samples_in[0::2]
28      samples_r =  samples_in[1::2]
29      samples_out[1::2] = 0.7 * samples_l # flip L & R, attenuate by 0.7
30      samples_out[0::2] = 0.5 * samples_r
31
32      data = np.chararray.tostring(samples_out) # convert back to string
33  #    data = wf.readframes(CHUNK) # read the next 1024 frames
34
35  stream.stop_stream() # pause audio stream
36  stream.close() # close audio stream
37
38  p.terminate() # close PyAudio & terminate PortAudio system
```

Lst. 9.5: Lesen, Bearbeiten, Abspielen und Speichern eines Audiofiles mit `wave` und `pyaudio`

Beim Arbeiten mit NumPy - Arrays sollte man achten auf:

Richtiges Format: Die Audiodaten im String sollten in NumPy Arrays mit 16 bit breiten Elementen (`np.int16`) umgewandelt werden

Überläufe: Bei numerischen Operationen kann man den 16 bit Wertebereich schnell verlassen - das hört sich nicht gut an ...

Vektorisierung: Die etwas kryptischen Zeilen 26 - 30 nutzen die schnelle Array-Arithmetik von Numpy, um die Samples von rechtem und linkem Kanal auseinander zu fieseln, zu bearbeiten und wieder zu einem Stream zusammenzufügen. Eine For-Schleife wäre um Größenordnungen langsamer!

Sounddevice

`Sounddevice` (http://python-sounddevice.rtfd.org/) ist ein weiterer Wrapper für die Portaudio-Library. Auf deren C-API greift man in Python über das „C Foreign Function Interface for Python“ (CFFI) zu, das daher ebenfalls installiert sein muss.

Da **`sounddevice`** reiner Pythoncode ist, kann man es installieren mit `pip install sounddevice`.

Wenn die Voraussetzungen gegeben sind, kann man numpy Arrays direkt abspielen:

```
1 import sounddevice as sd
2 sd.play(myarray, 44100)
```

Lst. 9.6: Beispiel für Sounddevice

PySoundCard

`PySoundcard` (https://github.com/bastibe/PySoundFile).

9.3.3. libsndfile mit Python-Wrappern

libsndfile (http://www.mega-nerd.com/libsndfile) ist eine in C geschriebene plattformunabhängige Library zum Lesen und Schreiben von Audiofiles (keine direkte Ausgabe) in diversen Fileformaten (u.a. wav, aiff, au, ircam, ogg and flac mit verschiedenen Wortlängen - siehe http://www.mega-nerd.com/libsndfile/#Features%22). Binaries sind für verschiedene Plattformen verfügbar.

Eine API für Python mit direktem Zugriff auf Numpy-Arrays bieten die folgenden Module:

PySoundfile

PySoundfile kann Audiodaten in verschiedenen Formaten lesen und schreiben. (https://github.com/bastibe/PySoundFile), das CFFI nutzt, um auf die C-API von `libsndfile` zuzugreifen.

Pysoundfile kann über `pip install ysoundfile` installiert werden, unter Anaconda verwendet man
`conda install -channel https://conda.anaconda.org/carlthome pysoundfile`

Beispiele unter http://bastibe.de/2013-11-27-audio-in-python.html.

pysndfile

pysndfile (https://forge.ircam.fr/p/pysndfile/) ist ein alternativer Wrapper, der Cython verwendet. Für eine pip-Installation muss daher der gcc - Compiler funktionsfähig eingebunden sein, für Anaconda gibt es keine vorkompilierten Binaries.

9.3.4. Python Wrapper für libsamplerate

Secret Rabbit Code (SRC) a.k.a. **libsamplerate** ist eine in C geschriebene Library von Erik de Castro Lopo (http://www.mega-nerd.com/SRC/) zum Resamplen von Daten in Numpy Arrays in hoher Qualität. Diese Library wird in zahlreichen Projekten verwendet bzw. nachgeahmt (z.B. von Audacity aufgrund der strikten Open-Source Lizenz). Implementiert sind drei bandbegrenzte sinc-Interpolatoren, die 97 dB SNR erreichen bei 97%, 90% bzw. 80% Bandbreite (zunehmende Konvertierungsgeschwindigkeit). Noch schneller sind der lineare und der Zero-Order Hold Interpolator bei deutlich schlechter Qualität. Im Test gegen andere Konverter unter http://src.infinitewave.ca/ schneidet die Library sehr gut ab. Die mathematischen Grundlagen sind beschrieben unter [Smi14] und [Smi11] und [PS97].

scikits.samplerate (http://scikits.appspot.com/samplerate) ist ein Python-Wrapper um `libsamplerate`, der als Binary auf Christoph Gohlkes Seite momentan allerdings nur für Python 2.x verfügbar steht. Der Wrapper stellt nur die einfache API zur Verfügung, auch die Dokumentation ist eher mager.

```
import numpy as np
import pylab as plt
from scikits.samplerate import resample

fs  = 44100.
fr  = 48000.
# Signal to resample
sins    = np.sin(2 * np.pi * 1000/fs * np.arange(0, fs * 2))
# Ideal resampled signal
idsin   = np.sin(2 * np.pi * 1000/fr * np.arange(0, fr * 2))

conv1   = resample(sins, fr/fs, 'linear')
conv3   = resample(sins, fr/fs, 'sinc_best')

err1    = conv1[fr:fr+2000] - idsin[fr:fr+2000]
err3    = conv3[fr:fr+2000] - idsin[fr:fr+2000]

plt.subplot(3, 1, 1)
plt.plot(idsin[fs:fs+2000])
plt.title('Resampler residual quality comparison')

plt.subplot(3, 1, 2)
plt.plot(err1)
plt.ylabel('Linear')

plt.subplot(3, 1, 3)
plt.plot(err3)
plt.ylabel('Sinc')

plt.savefig('example1.png', dpi = 100)
```

Lst. 9.7: Samplerate-Conversion mit scikits.samplerate

Ein weiterer Wrapper ist **PyLibSampleRate** (https://code.google.com/p/pyzic/wiki/PyLibSampleRate), der ebenfalls nur die Grundfunktionalität von libsamplerate zur Verfügung stellt und die DLL über `ctypes` (Kap. C.5) einbindet.

Leider gibt es keine Binaries für `libsamplerate`, man muss sie selbst erstellen. Die einzige Abhängigkeit besteht zur `libsndfile` (siehe oben). Um `libsamplerate.dll` unter Win32 mit MS Visual C++ zu erstellen, benötigt man die folgenden Schritte (http://www.mega-nerd.com/SRC/win32.html):

1. Using WinZip in the GUI, open the `libsamplerate-0.X.Y.tar.gz` file and extract the files into a directory. The following example assumes `C:\` .
2. In the directory containing the extracted files, find the file `Win32\Makefile.msvc` and open it in a text editor (ie Notepad or similar).
3. Find the line which starts with MSVCDir and modify the directory path to point to the location of MSVC++ on your machine. This allows the makefile to inform the compiler of the location of the standard header files.
4. Copy libsndfile-1.dll, libsndfile-1.lib and libsndfile-1.def from the directory libsndfile was installed in to the the directory containing libsamplerate.
5. Copy the header file include/sndfile.h from where libsndfile was installed to the Win32 directory under the libsamplerate directory.

6. Open a Command Shell and cd into the `libsamplerate-0.X.Y` directory.

7. Make sure that the program nmake (which is part of the MSCV++ package) is in a directory which is part of your PATH variable.

8. Type in the command

   ```
   C:\libsamplerate-0.X.Y> make
   ```

   and press <return>. You should now see a a large number of compile commands as libsamplerate.dll is built.

9. To check that the built DLL has been compiled correctly type in and run the command

   ```
   C:\libsamplerate-0.X.Y> make check
   ```

which will compile a set of test programs and run them. If any of the programs fail the error message will be help in debugging the problem. (Note that some of the tests require libsndfile or libfftw/librfftw and are not able to run on Win32).

Wenn alles glatt gelaufen ist, findet man die DLL `libsamplerate.dll` und einen LIB File `libsamplerate.lib` im aktuellen Directory. Man muss jetzt nur noch diese beiden Files zusammen mit dem Headerfile `src/samplerate.h` in das Projekt kopieren, in dem `libsamplerate` eingesetzt werden soll.

9.3.5. ... und was ist mit mp3?

Aus Lizenzgründen kann man mit den meisten Python-Distributionen keine mp3-Files abspielen. Ein Workaround ist, ein externes Programm wie Lame für die Konvertierung aufzurufen:

```
import subprocess, os
from array import array as raw_array

class Mp3Reader:
    '''Reads samples from stereo  mp3 file'''
    def __init__(self,fname, channels=2):
        self.dec=subprocess.Popen(("lame", "--decode", "-t", "--quiet",
                                   mp3file, "-"), stdout=subprocess.PIPE)
        self.stream=self.dec.stdout
        self.channels=channels

    def read(self,n):
        bytes=n*2*self.channels
        input=self.stream.read(bytes)
        if not input:
            return
        a = raw_array('h')
        a.fromstring(input)
        a=array(a)
        if self.channels>1:
            a.shape=-1,self.channels
            a=np.mean(a,1)
        return a
r=Mp3Reader(mp3file,2)
```

9.3.6. Andere

audiolazy (https://github.com/danilobellini/audiolazy) ist ein Real-Time Expressive Digital Signal Processing (DSP) Package für Python; Daten werden erst dann generiert wenn sie auch wirklich gebraucht werden. Integration mit PyAudio, Numpy, Matplotlib, ...

pyo (http://code.google.com/p/pyo/) ist ein C geschriebenes Python Modul mit einer Vielzahl von Klassen und Objekten zur kreativen Real-Time Audio Bearbeitung und Klangerzeugung. Leider bis jetzt nur für Python 2.x.

aubio (http://aubio.org/) ist eine Bibliothek, um in Real-Time High-Level Merkmale aus Musik zu extrahieren wie Tonhöhe, Tempo, Anschläge etc.

librosa (http://bmcfee.github.io/librosa) hat die gleiche Zielsetzung, mit sehr umfangreicher Funktionalität

Essentia (http://essentia.upf.edu/) ist eine Open-Source C++ library für Audioanalyse und audiobasierte Informationsgewinnung

pygame (s.u.) ist eine Bibliothek zur Erstellung von Spielen mit vielfältigen Optionen für Mehrkanalsound.

phonon als Teil der Qt-Library bietet Soundunterstützung für Qt-GUIs.

Python Audio Effects GUI (https://sites.google.com/site/ldpyproject) ist eine Demo-Anwendung von `scikits.audiolab` (s.o.), die gleichzeitig zeigt, wie man GUIs aufbaut und Code in übersichtliche Module aufteilt.

Pitchshifting (http://zulko.github.io/blog/2014/03/29/soundstretching-and-pitch-shifting-in-python/) mit Python in Real-Time

Audioprogramming (http://audioprograming.wordpress.com/) beschreibt u.a. einen Vocoder und einen Pitchshifter / Time-stretcher in Python

9.4. Bildverarbeitung

Die Bibliothek **scipy.ndimage** enthält bereits zahlreiche Funktionen zur multidimensionalen Bildverarbeitung.

Die Python Imaging Library **Pillow** (**https://pillow.readthedocs.org/**) ermöglicht einfachere Bildverarbeitungsaufgaben (Anzeigen, Lesen, Schreiben, Formatkonvertierungen, ...).

Mahotas (http://luispedro.org/software/mahotas) ist eine Sammmlung von Funktionen zur Bildbearbeitung und -erkennung, geschwindigkeitskritische Routinen sind in C++ geschrieben und verwendet ausschließlich Numpy Arrays als Datentyp. **scikits.image** (http://scikit-image.org/) verfolgt einen ähnlichen Ansatz.

Die **OpenCV** (http://opencv.org/) Library hat auch eine Python-API.

9.5. Mess- und Regelungstechnik

Python eignet sich sehr gut zur Messautomatisierung und wird daher in öffentlichen Forschungseinrichtungen und auch in der Industrie oft hierfür eingesetzt. Gründe hierfür sind die leichte Einbindung von DLLs / C-Code, gute Unterstützung von Interfaces (pySerial, pyVISA, pyParallel) und die leistungsfähigen Tools zur Datenvisualisierung.

Ein gutes Buch hierzu ist John M. Hughes, „Real World Instrumentation with Python - Automated Data Acquisition and Control Systems“ (http://shop.oreilly.com/product/9780596809577.do).

Die Library **scipy.signal** enthält bereits viele Funktionen zur Modellierung und Analyse regelungstechnischer Systeme, speziellere Funktionen enthält das Modul **python-control** (http://sourceforge.net/projects/python-control/), das am California Institute of Technology (http://www.cds.caltech.edu/~murray/wiki/index.php/Python-control) entwickelt wurde und wird. Mit dem Untermodul `control.matlab` werden viele Befehle der Matlab Control Toolbox nachbildet.

Tango (http://www.tango-controls.org/) ist ein Großprojekt zur Entwicklung von objektorientierten Steuerungssystemen, das von verschiedenen Forschungsinstitutionen entwickelt wird. Man kann Tango ohne aufwändige Installation als Image (Tango Box Virtual Machine) herunterladen und in einer Virtual Machine ausführen. Das Video unter http://vimeo.com/79554181 gibt einen kurzen Überblick über Tango und TangoBoxVM.

9.6. Physik und Spiele

9.6.1. Physics

Dieses Modul ergänzt Python um physikalische Konstanten und Einheiten: Beispiel aus Anders Lehmann, „Using IPython in the Classroom“, EuroScipy 2013http://www.youtube.com/watch?v=0_vd9ctwQW4:

```
import physics
R = 10. cm
I = 1. A
B = mu0 * I / (2 * R) # Anteil des kreisförmigen Leiters
B += mu0 * I / (2 * pi * R) # Anteil des unendlich ausgedehnten Leiters
B.convert('T') # Umwandlung von Basis-Einheiten in Tesla
print 'Die magnetische Feldstärke im Zentrum ist B = {:.2E}"'.format{B}
```

9.6.2. Agros2D

Agros2D ist eine 2D Multiphysics FEM Software (http://www.agros2d.org), die sich mit Python objektorientiert skripten lässt.

9.6.3. pyGame

Pygame (http://www.pygame.org) ist eine Bibliothek, die Grafik- und Soundunterstützung für einfache Spiele zur Verfügung stellt. Zusammen mit der Bibliothek pymunk für die 2D Physik starrer Körper eignet sie sich auch hervorragend, um physikalische Probleme zu animieren (http://www.petercollingridge.co.uk/pygame-physics-simulation).

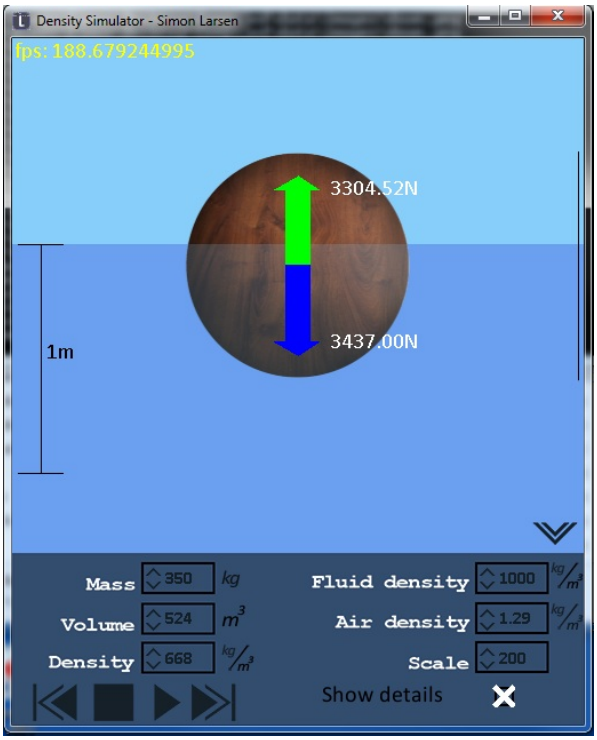


Abb. 9.3.: Screenshot aus Density Simulator [http://lagusan.com/density-simulator/]

9.6.4. VPython

Für die Visualisierung von physikalischen Zusammenhängen wird oft die 3D-Bibliothek **VPython** eingesetzt - siehe unten.

9.7. 3D-Grafik

9.7.1. Matplotlib

Auch die Matplotlib bietet bereits eine Vielzahl an 3D-Plotting-Optionen (`plot, scatter, mesh, surf, contour, ...`), die viele Anwendungsfälle abdecken. Nachteilig sind:

Fehlende Hardwarebeschleunigung: Bei Plots mit vielen Datenpunkten, Lichtquellen und Tiefenstaffelung (z.B. `surf`) ist das Drehen und Zoomen des Plots sehr langsam. Alternativ kann man Winkel und Zoom direkt beim Plotten angeben.

Fehlende Funktionalität:

9.7.2. Mayavi

Mayavi http://code.enthought.com/projects/mayavi/ und http://docs.enthought.com/mayavi/mayavi/ ist die High-End Grafik-Engine für 3D Visualisierung und Animation von Daten. Basis von Mayavi ist das Visualisation Toolkit (VTK, http://www.vtk.org/).

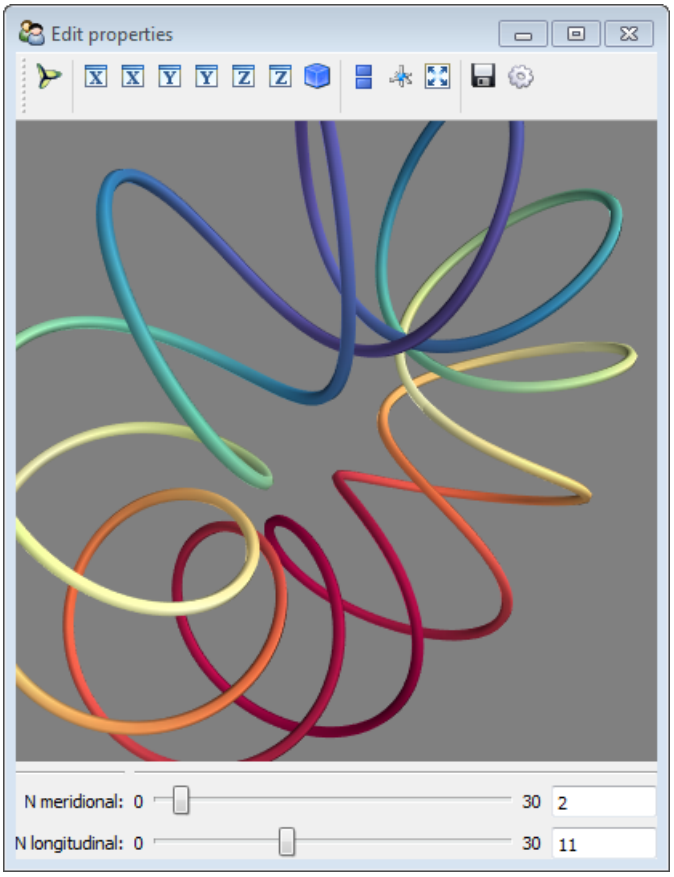


Abb. 9.4.: Interaktive 3D-Visualisierung mit Mayavi und Qt

WinPython muss für die Installation von Mayavi in der Registry eingetragen sein (mit dem WinPython Control Panel), danach kann man das passende vorkompilierte ETS („Enthought) Modul von Christoph Gohlke (http://www.lfd.uci.edu/~gohlke/pythonlibs/) über das WinPython Control Panel installieren. Außerdem muss `ConfigObj` z.B. von PyPI nachinstalliert werden. Die Enthought und Anaconda - Installation enthalten bereits Mayavi. Bei Spyder müssen die in Abb. 9.5 gezeigten Einstellungen vorgenommen werden. Viele Mayavi-Beispiele im Internet wurden unter Unix erstellt, manche davon stürzen unter Windows und OS X ab. Ein Grund dafür ist ein fehlerhafter Default für die maximale Anzahl von Datenpunkten für „Mask input points“ (http://blog.seljebu.no/2013/11/python-segfault-on-mayavi-masking/). Workaround: Manuelles Setzen der Anzahl der Punkte (Zeile 7 in Listing 9.8).

```
1 # Plot figure with white background
2 fig = figure(1, size=(400, 400), bgcolor=(1, 1, 1), fgcolor=(0, 0, 0))
3 quiver3d(x,y,z,bx,by,bz)
4 #
5 vectors = fig.children[0].children[0].children[0]
6 # "Manually" set maximum number of points:
```

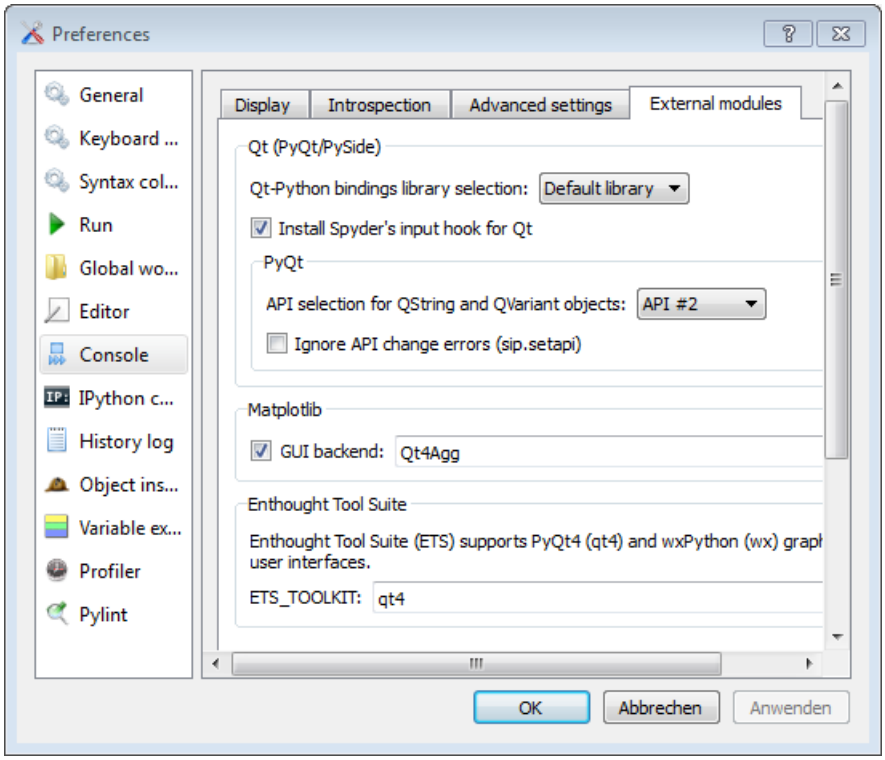


Abb. 9.5.: Spyder Setup für Mayavi

```
7 vectors.glyph.mask_points.maximum_number_of_points = 1000
8 vectors.glyph.mask_input_points = True # turn masking on
```

Lst. 9.8: Manuelles Setzen der max. zu maskierenden Punkte bei Mayavi um einen Crash (malloc too large) bei OS X und Windows zu vermeiden

Mayavi - Visualisierungen können auch in Qt eingebettet werden und so interaktiv verwendet werden (Abb. 9.4).

9.7.3. VPython

VPython (http://vpython.org/) ist eine Umgebung zur Erzeugung von interaktiven 3D-Visualisierungen und Animationen, basierend auf einfachen geometrischen Objekten. Transparenz, Schatten und Oberflächen werden unterstützt.

VPython setzt auf der wxWidgets Bibliothek auf, die man für Python über die Module wxPython und wxPython-common erhält (z.B. von Christoph Gohlkes Homepage) - wxWidgets muss *vor* VPython installiert werden, und zwar Version 2.8 für VPython 5.7 und Version 2.9 für VPython 6.0. Listing 9.9 zeigt den vollständigen Code zur Darstellung eines kubischen Kristallgitters und Abb. 9.6 den resultierenden 3D-Plot.

```
1 from visual import sphere, display, color
2
3 d = display(x=0,y=0,height=400,width=400, background = color.white, title='Kubisches
      Kristallgitter')
```

```
4 L = 3
5 R = 0.3
6 for i in range(-L,L+1):
7     for j in range(-L,L+1):
8         for k in range(-L,L+1):
9             sphere(pos=[i,j,k],radius=R, color=color.red)
```

Lst. 9.9: Visualisierung eines kubischen Kristallgitters mit VPython

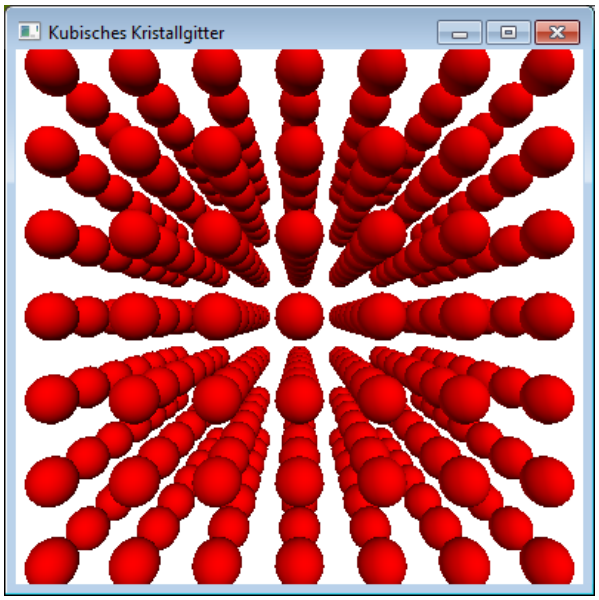


Abb. 9.6.: Visualisierung eines kubischen Kristallgitters mit VPython

Das Beispiel stammt aus dem Buch *Computational Physics* von Mark Newman, http://www-personal.umich.edu/~mejn/computational-physics/ aus dem (als PDF herunterladbaren) Kapitel „Graphics and Visualization“.

Das Lehrbuch *Matter & Interactions* von Ruth Chabay und Bruce Sherwood (http://matterandinteractions.org) setzt VPython zur Visualisierung ein:

> VPython is an unusually easy to use 3D programming environment that is based on the widely-used Python programming language. VPython can be used by students with no previous programming experience, which makes it feasible to include computational modeling in the Matter & Interactions curriculum.

Alle Python-Programme des Buchs können von der Homepage heruntergeladen werden.

Interaktive und animierte 3D-Graphen lassen sich ebenfalls erzeugen, sehr eindrucksvoll anzuschauen sind z.B. die Animationen von quantenmechanischen Wellenfunktionen auf http://www.youtube.com/watch?v=imdFhDbWDyM !

9.8. Microcontroller und andere Hardware

9.8.1. myHDL

myHDL (http://myhdl.org) erweitert Python um Optionen zur Simulation von digitalen Systemen. Besonders interessant ist, dass der Code automatisch zu VHDL / Verilog konvertiert werden kann und sich damit auch für die Synthese von ASICs und FPGAs eignet!

Jan Decaluwe (PyConTw 2013): http://www.youtube.com/watch?v=LSgOpvr8FII zum Konzept von myHDL

Christopher Felton (PyOhio 2013): http://www.youtube.com/watch?v=949JjIK62Lg mit mehr Beispielen

9.8.2. MiGen / MiSOC

Ähnlich wie myHDL ist migen (https://github.com/m-labs/migen) „a Python toolbox for building complex digital hardware“. Mit Migen wurde ein synthesisierbares System-On-Chip realisiert, miSOC (https://github.com/m-labs/misoc).

9.8.3. MicroPython

microPython (http://micropython.org/) ist eine schlanke und schnelle Implementierung von Python 3, die für Microcontroller optimiert wurde. Der volle Sprachumfang von Python 3 wird unterstützt, aber nur ein Teil der Standard Libraries.

Über eine Kickstarter-Kampagne (https://www.kickstarter.com/projects/214379695/micro-python-python-for-microcontrollers) wurde das MicroPython Board finanziert. Es basiert auf einem STM32F405RG Microcontroller mit 168 MHz Cortex-M4F CPU (32-bit Hardware Floating Point Unit, 1 MB Flash, 192 kB RAM) und soll ab April 2014 erhältlich sein. Source Code wird über USB auf das Board kopiert, dort zu Byte- oder Maschinencode (selektierbar für jede Funktion) kompiliert und abgearbeitet. Wer hier an eine „BASIC Briefmarke“ mit Python anstatt Basic und mehr Wumms denkt, liegt vermutlich nicht ganz falsch. MicroPython behauptet, die Garbage Collection deutlich intelligenter implementiert zu haben als z.B. pyMite und so Interrupts und andere zeitkritische Routinen zuverlässiger auszuführen. Ein weiterer Unterschied ist, dass MicroPython nativen Maschinencode erzeugen kann, auch Inline-Assembleranweisungen sind möglich.

Auch für das HydraBus-Board (http://hydrabus.com/hydrabus-1-0-specifications/), das ebenfalls auf einer STM Cortex-M4F CPU basiert, gibt es eine microPython Implementierung.

9.8.4. pyMite

Python-on-a-Chip (p14p, http://code.google.com/p/python-on-a-chip/) ist eine reduzierte Python Virtual Machine (Codename: PyMite), die einen großen Teil des Python-Sprachstandards auf uCs ohne Betriebssystem unterstützt. Außerdem gehören Treiber, High-Level Libraries und verschiedene Tools zu p14p. Implementiert wurde pyMite bis jetzt u.a. auf einem Arduino Mega und einem Microchip PIC 24.

9.8.5. pyMCU

Das pyMCU-Projekt (http://www.circuitsforfun.com/pymcu.html) verfolgt einen anderen Ansatz als MicroPython und pyMITE: pyMCU ist ein „Python Controlled Microcontroller“ (Microchip PIC uC 16F1939). Der Python-Programmcode läuft auf dem PC und steuert über den USB-Bus den uC als Slave Device an. Der uC kann dann z.B. eine LED einschalten oder Werte über die integrierten A/D-Wandler einlesen. Mit dem Python Modul pyMCU können dann die Analog- und Digital-I/Os auf einfache Weise angesprochen werden.

A. ZTEX-FPGA Board

In diesem Kurs wird das FPGA-Board 2.01 der Firma ZTEX verwendet, das mit einem Xilinx Spartan 6 FPGA XC6SLX16 (optional: XC6SLX25) bestückt ist.

Hauptvorteile des Boards sind

Bus Powered: Das Board kann über den USB-Bus mit Strom versorgt werden

USB - programmierbar: Das FPGA kann ohne zusätzliches Programmiergerät bequem über die USB-Buchse mit Hilfe des Open Source SDK programmiert werden.

SDK:

und einem Cypress EZ-USB FX2 Microcontroller (CY7C68013A) zur Kommunikation und zum Programmieren des FPGAs.

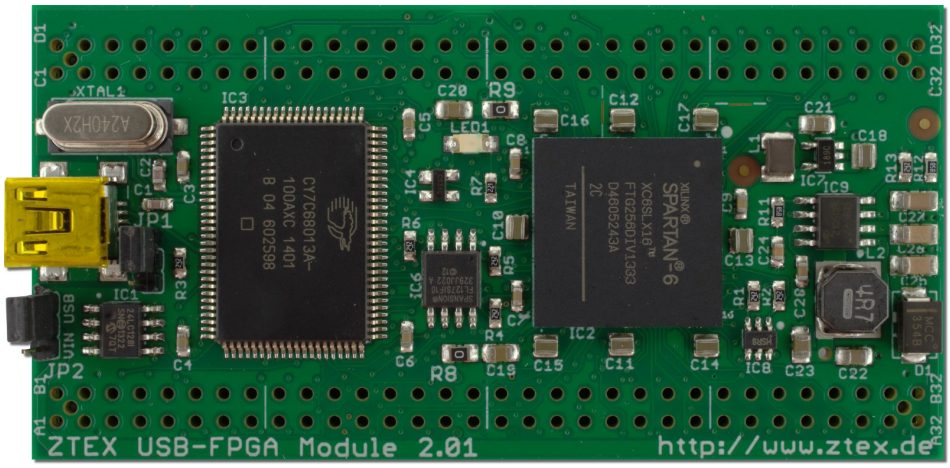


Abb. A.1.: Foto ZTEX USB-FPGA Modul 2.01 [http://www.ztex.de]

A.1. Hardware

Das Board ist ausführlich beschreiben unter http://www.ztex.de/usb-fpga-2/usb-fpga-2.01.e.html. Das Blockdiagramm Abb. A.2 zeigt, dass schnelle Kommunikation und die Programmierung des FPGAs über das USB 2.0 Interface des Cypress FX2 Microcontrollers laufen. Die Firmware des uC und der FPGA Programmierfile sind in separaten EEPROMS abgelegt.

Jumper 1: Wähle zwischen Bus-Powered und externer Versorgung

Jumper 2: Wähle zwischen zwei I2C-Bus Adressen

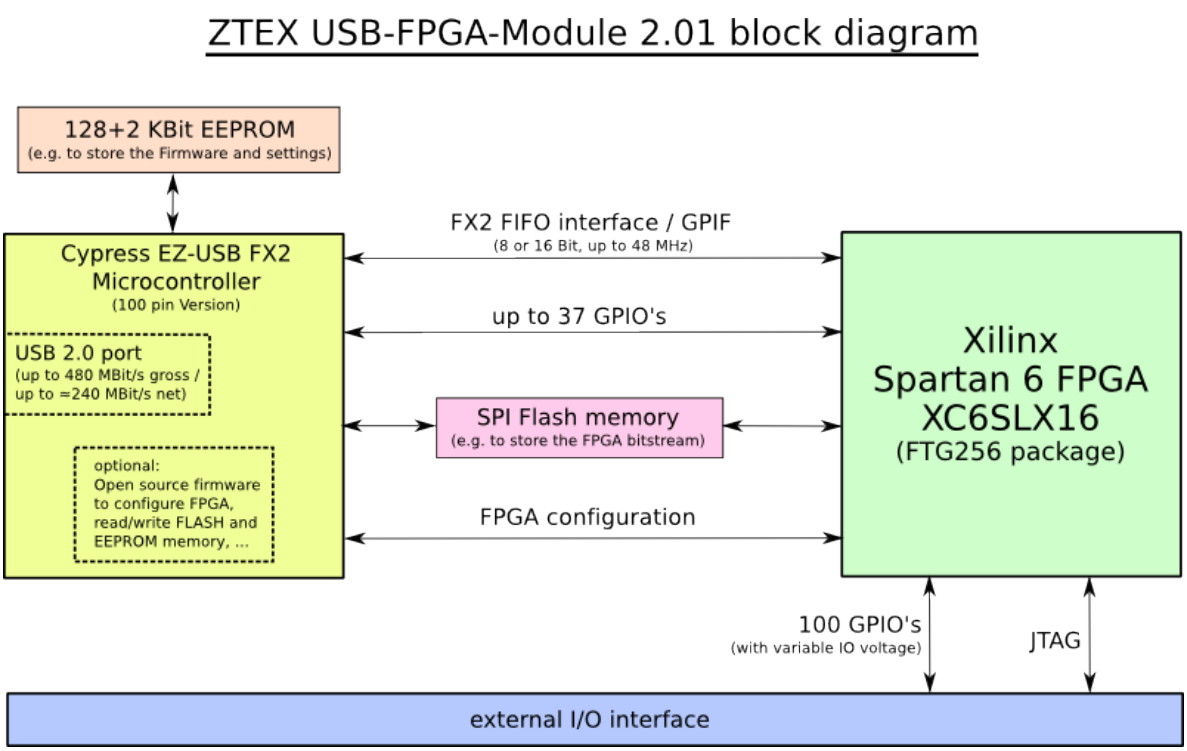


Abb. A.2.: Blockschaltbild des ZTEX USB-FPGA Moduls [http://www.ztex.de]

Leuchtdiode: Grün leuchtend heißt, das FPGA ist unkonfiguriert

A.2. Software

Um das Board zu programmieren und um damit zu kommunizieren, benötigt man das ZTEX Software Development Kit (SDK) http://www.ztex.de/firmware-kit/. Das SDK besteht im Wesentlichen aus zwei Teilen:

Firmware Kit: Zum Kompilieren der Firmware für den EZ-USB FX2 uC (von uns nicht benötigt)

Host Software API:

Java Host Software API:

- Plattformunabhängig (geschrieben in Java), Softwarepakete in einem .jar-File
- Direkter Firmware upload zum EZ-USB FX2 uC (EEPROM)
- Direkter Zugriff zum SPI-Flash, Bitstream Upload
- Direkter Bitstream upload zum FPGA - Konfigurations RAM
- FPGA lädt automatisch den Bitstrem aus dem SPI-Flash beim Power-Up
- MAC-EEPROM support, used to store configuration data (?)

DeviceServer (G)UI:

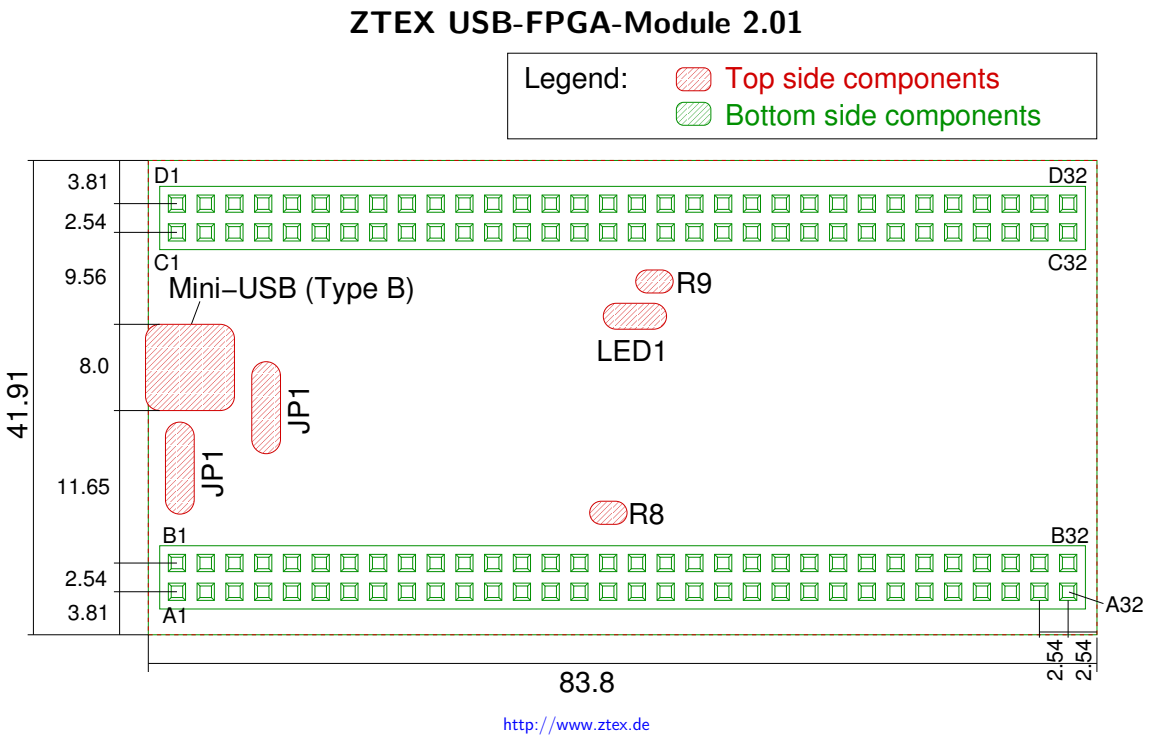


Abb. A.3.: Zeichnung ZTEX USB-FPGA Modul 2.01 [http://www.ztex.de]

- DeviceServer is an easy-to-use user interface for ZTEX Boards
- Two access methods: HTTP (using web browser) and a simple socket protocol
- Supports uploading of Firmware and Bitstream into volatile and non-volatile memory
- Supports simple I/O: read and write to Endpoints

FWLoader utility:

- Command line utility to upload Firmware and Bitstream
- Supports all upload methods and targets of the API / Firmware Kit

A.2.1. Installation

Voraussetzung: Das Java Runtime Environment JRE 6 oder höher muss installiert sein. Das kann man im Terminal mit `java -version` überprüfen.

Beispiele zu Installation und Nutzung finden Sie unter http://wiki.ztex.de/doku.php.

1. Download und Auspacken der Software `ztex-160129.zip`
2. Treiberinstallation: Beim Anstecken unter Windows muss man den Treiber `ztex.inf` aus dem Unterverzeichnis `libusb-win32` händisch installieren. Unter Linux kann das Modul direkt mit Hilfe der `libusb` Library angesprochen werden.

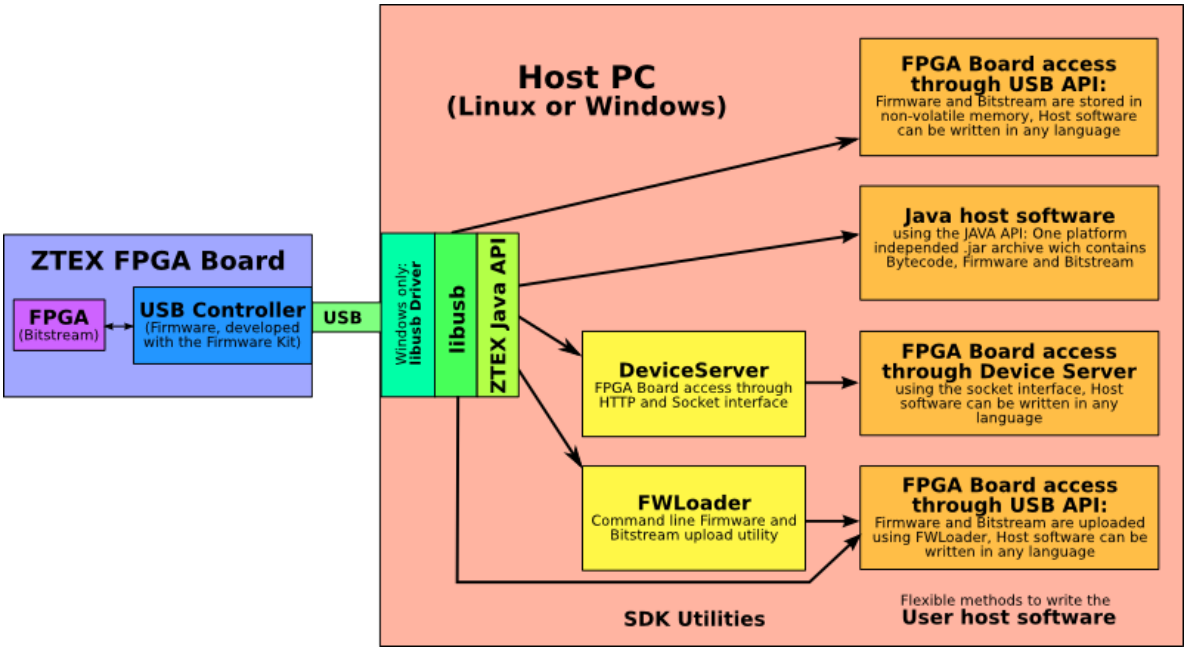


Abb. A.4.: Komponenten des ZTEX SDK [http://www.ztex.de]

3. Testen mit einem Beispiel: Im Unterverzeichnis `examples\usb-fpga-2.01` findet sich u.a. das Beispiel `ucecho`, das man per Batchfile `ucecho.bat` starten kann. Er öffnet ein Terminal, schickt einen von Ihnen eingegebenen String zum FPGA, der ihn in Großbuchstaben umwandelt und zurückschickt.

A.2.2. Eigene Java-Archive erstellen

http://de.wikihow.com/Eine-.Jar-Java-Datei-ausf%C3%BChren

A.2.3. Kommunikation mit Python

Voraussetzung: Installation von PyUSB (http://walac.github.io/pyusb/)

A.3. FPGA-Designsoftware

```
1
2 vectors = fig.children[0].children[0].children[0]
```

Lst. A.1: BlaBlubb

A.3.1. Installation

Die Anforderungen für die verwendete Xilinx-Designsoftware sind:

- mindestens 8 GB RAM
- mindestens 20 GB freier Plattenplatz
- Windows oder Linux

Download: Laden Sie zunächst die zu Ihrem Betriebssystem passende Designsoftware von `http://www.xilinx.com/support/download.html` herunter. Achtung: laden Sie nicht die Vivado Umgebung herunter - die funktioniert nur mit FPGAs ab der 7er Serie (Artix, Kintex, Zynq, ...). Das von uns verwendete FPGA-Board enthält einen Spartan-6 FPGA.

Wechseln Sie zum „ISE“ Tab, nehmen Sie sich Zeit und eine dicke Leitung - am besten laden Sie die knapp 8 GB mit dem „Split Installer“ in vier Paketen herunter und extrahieren Sie die Files.

License Agreement: Akzeptieren Sie die Licence Agreements für die Xilinx-Tools, den Schnüffelservice WebTalk (verpflichtend für die kostenlose Version) und die Open Source Tools.

Edition: Wählen Sie dann für die Installation die **ISE WebPack Edition** aus, das ist das einzige kostenlose Paket in der Auswahl, das für unsere Zwecke aber ausreichend ist.

Installationsoptionen: Lassen Sie alle Optionen aktiviert, lediglich die Unterstützung für die schnelle Ethernet Cosimulation (WinPCap Driver) benötigen wir nicht.

Installationsverzeichnis: Wo immer es Ihnen gefällt, Sie benötigen ca. 17,5 GB Plattenplatz.

Lizenz: Am Ende der Installation (ca. 20 ... 30 min.) öffnet sich der „Xilinx License Configuration Manager“ und Sie werden nach der gewünschten Lizenz gefragt: Wählen Sie die Option „Get Free Vivado/ISE WebPack License“. Wenn Sie noch nicht registriert sind, unterstützt Sie Xilinx gerne beim Aufnehmen Ihrer Daten :-). Auf der Xilinx Web Site erzeugen Sie dann Ihren Lizenzfile, indem Sie „ISE WebPack License“ auswählen und „Generate Node-Locked License“ anklicken. Damit erzeugen Sie eine lokale Lizenz für Ihren PC (im Gegensatz zu einer floating License, die man bei Bedarf in einem Hochschul- oder Firmennetz auschecken und wieder zurückgeben kann). Sie können wählen, ob die Lizenz an die ID Ihrer Festplatte oder an eine MAC-Adresse gebunden werden soll.

Im „Configuration Manager“ Fenster (das hoffentlich noch offen ist) können Sie ggf. nachschauen, welche IDs Ihrer Rechner hat. Einmal noch das License Agreement abnicken, dann wird das Lizenzfile an die hinterlegte eMail Adresse geschickt. In der Mail finden Sie ausführliche Anweisungen, letzten Endes müssen Sie aber nur das angehängte `*.lic` File abspeichern und dem „Xilinx License Configuration Manager“ übergeben.

Umgebungsvariablen: Setzen Sie die Umgebungsvariablen durch Ausführen von `<INSTALL_DIR>/14.7/ISE_DS/s`

A.3.2. Starten

B. Erzeugen von Binaries

Um Python - Programme an Kunden (oder Studenten :-)) weiterzugeben ohne eine Pythoninstallation beim Anwender vorauszusetzen, kann man sie in Binaries umwandeln - und hat bei Python wie so oft die Qual der Wahl:

Die bekanntesten Projekte sind:

pyinstaller: http://www.pyinstaller.org/, Cross-Platform, wird aktiv weiterentwickelt, Python 2.7 und Python ≥ 3 3 werden unterstützt. Sehr gute Dokumentation. Details s.u.

cx_Freeze: http://cx-freeze.sourceforge.net/, Cross-Platform, wird aktiv weiterentwickelt, unterstützt Python 2.x und 3.x, aber keine Option für Single-File Installer (Workaround: Separater Installer wie InnoSetup oder upx). Ist in WinPython enthalten. Details s.u.

py2exe: http://sourceforge.net/projects/py2exe/, nur für Windows, wird anscheinend nicht mehr weiterentwickelt (letzte Version: Mai 2013)

bbFreeze: https://pypi.python.org/pypi/bbfreeze/, Windows und Unix, keine Weiterentwicklung seit Anfang 2014, unterstützt nur Python 2.x. Nur wenig Dokumentation.

pyqtdeploy: https://www.riverbankcomputing.com/software/pyqtdeploy/intro, Cross-Platform, erstellt pyQt Anwendungen für die üblichen Betriebssysteme sowie für Android, iOS und WindowsRT. Python und Qt/pyQt-Module werden zu einem single-File Executable kompiliert.

Freeze-Utilities binden üblicherweise alle Untermodule von Qt, Numpy etc. ein, auch wenn vielleicht nur ein Bruchteil benötigt wird, so dass man meist Executables $>$ 100 MB erhält. Durch manuelles Ausschließen von Binaries kann man die Größe der resultierenden Executables oft deutlich verringern. Mit **`ldd -u`** unter Linux, **`Dependency Walker`** (http://dependencywalker.com/) oder **`ListDLLs`** (https://technet.microsoft.com/en-us/sysinternals/bb896656.aspx) unter Windows lassen sich nicht benötigte DLLs / so's finden. Alternativ: Ausprobieren ...

B.1. PyInstaller

PyInstaller lässt sich leicht downloaden und installieren mit `pip install pyinstaller`.

Video: https://www.youtube.com/watch?v=11Q2QADsAEE.

Wenn alles glatt läuft, ist auch die Anwendung einfach: Mit

```
pyinstaller <pfad-zu>\<mein_programm>.py
```

wird der Build-Prozess gestartet. Nach Ablauf des Build-Prozesses ist man um das Directory `<mein_programm>` mit den beiden Subdirectories `build` und `dist` reicher.

Wenn alles glatt durchgelaufen ist, kann das `build` Subdirectory gelöscht werden. Wenn nicht, findet man hier u.a. den File `warn<PROJECT>.txt` mit Warnungen und Fehlern. Das `dist` Subdirectory enthält alles, was der Endanwender braucht. Dieses Directory kann gezipped und weitergegeben werden, der Nutzer muss das Archiv dann entpacken und `meinfile.exe` ausführen. Übersichtlicher ist die „one-file Option“, mit der alles in ein selbst-extrahierendes Archiv gepackt wird:

```
pyinstaller <pfad-zu>\meinfile.py --onefile
```

Das Archiv wird beim Aufruf in ein temporäres Directory mit zufälligem Namen ausgepackt umd ausgeführt; nach Beendigung wird das Temp-Directory wieder gelöscht sofern das Programm nicht „abgeschossen“wurde.

Beim ersten Aufruf legt pyinstaller ein File `<meinfile>.spec` an, in dem Optionen, Pfade etc. abgelegt sind. Damit muss man beim nächsten Start einfach nur

```
pyinstaller meinfile.spec
```

angeben. Dieser File ist im Python-Format und darf auch Python - Befehle enthalten. Listing B.1 zeigt ein Beispiel für einen `*.spec` File mit manuellen Ergänzungen.

Nach der Analyse-Phase sind alle Binaries, die pyInstaller für notwendig erachtet, in `a.binaries` aufgelistet, einem Set von Tuples im Format (Filename, Filepath, Typcode). `*.pyd` - Files sind vergleichbar mit einer `*.dll`, der Suffix muss aber nicht angegeben werden. Indem man über `a.binaries` iteriert, können auch ganze Module ausgeschlossen werden. Ob das Executable dann noch funktioniert, ist allerdings eine andere Frage ... Analog dazu können auch fehlende DLLs hinzugefügt werden.

```
1  # -*- mode: python -*-
2  a = Analysis(['..\\..\\digamp\\verstaerker.py'],
3               pathex=['D:\\Daten\\design\\python\\Beike\\digamp'],
4               hiddenimports=[],
5               hookspath=None,
6               runtime_hooks=None)
7  # Exclude complete modules by iterating over the TOC:
8  #a.binaries = [x for x in a.binaries if not x[0].startswith("numpy.linalg")]
9  # Exclude specific binaries:
10 a.binaries = a.binaries - TOC([
11  ('QtNetwork4.dll', '', ''),
12  ('_ssl', '', ''),
13  ('numpy.core._dotblas', '', ''),
14  ])
15 # Add a single missing dll...
16 #a.binaries = a.binaries + [
17 #  ('opencv_ffmpeg245_64.dll', 'C:\\Python27\\opencv_ffmpeg245_64.dll', 'BINARY')]
18 pyz = PYZ(a.pure)
19 exe = EXE(pyz,
20           a.scripts,
21           a.binaries,
22           a.zipfiles,
23           a.datas,
24           name='verstaerker.exe',
25           debug=False,
26           strip=None,
27           icon='Logo_LST_4.ico',
```

```
28          upx=True,
29          console=True )
```

Lst. B.1: Beispiel für einen *.spec - File

B.2. cx_Freeze

Der grundlegende Aufruf ist ganz einfach:

```
cxfreeze myPythonFile.py --target-dir=<mydir> --include-module=atexit,numpy
```

Es wird das Zieldirectory `mydir` angelegt bzw. überschrieben, das den File `myPythonFile.exe` enthält und diverse Libraries (`*.pyd` und `*.dll`). Leider ist cx_Freeze nicht schlau genug, um nur die Libraries einzubinden, die wirklich gebraucht werden - hier ist die import-Liste maßgeblich (siehe auch http://stackoverflow.com/questions/27281317/cx-freeze-preventing-including-unneeded-packages).

Um alle Files in ein Executable zu bündeln, kann z.B. InnoSetup (http://www.jrsoftware.org/isinfo.php) verwendet werden.

Eine schöne Einführung mit Beispiel gibt https://www.smallsurething.com/a-really-simple-guide-to-packaging-your-pyqt-application-with-cx_freeze/.

B.3. pyqtdeploy

pyqtdeploy wird von Riverbank Computing entwickelt und mit BSD-Lizenz abgegeben.

B.4. WinPython als Minimalsystem

Eine weitere interessante Variante wird von Cyrille Rossant beschrieben (http://cyrille.rossant.net/tag/winpython/): Man erzeuge ein Directory mit dem Python Programm und einer Winpython-Distribution, die auf das absolute Minimum abgespeckt wurde und packe das Ganze mit dem Windows-Installer InnoSetup (http://www.jrsoftware.org/isinfo.php).

C. Need for Speed - das muss kacheln!

Python als interpretierter Sprache hängt der Ruf nach, (zu) langsam zu sein. In manchen Fällen trifft das sicherlich zu, in vielen Fällen ist es aber „schnell genug“. Falls nicht, gibt es verschiedene „Nachbrenner“ - siehe auch http://www.scipy.org/PerformancePython oder http://technicaldiscovery.blogspot.de/2011/06/speeding-up-python-numpy-cython-and.html), Suchstichwort „High Performance Python“, z.B. das Buch „High Performance Python“ von Micha Gorelick und Ian Ozsvald, O'Reilly Verlag, 2014.

Eine gute Diskussion zu „Python ist langsam“ gibt es unter http://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/.

Zunächst aber gibt es ein paar Do's und Don'ts in Mark Litwintschiks „Faster Python“: http://tech.marksblogg.com/faster-python.html.

C.1. Benchmarking

Für ein Benchmarking muss man erst einmal messen, wie lange unterschiedliche Implementierungen benötigen:

C.1.1. `timeit`

`timeit` als Python Standardmodul umgeht einige Fallen, die bei der einfachen Messung der Zeit auftreten können (garbage collection etc.). Achtung: `timeit` misst die verstrichene Zeit („Wall clock time“ = Wanduhr), nicht CPU Zyklen!

Das Modul kann von der Kommandozeile ausgeführt werden:

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop

$ python -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 25.2 usec per loop
```

Lst. C.1: Laufzeitmessung mit `timeit` von der Kommandozeile

Die Anzahl der benötigten Schleifendurchläufe wird hier automatisch bestimmt.

In IPython ist `timeit` als Magic implementiert:

```
%timeit "-".join(map(str, range(100)))
```

Lst. C.2: Laufzeitmessung mit `timeit` unter IPython

Schließlich kann man `timeit` auch innerhalb von Pythonskripten anwenden:

```
import timeit
timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
```

Lst. C.3: Laufzeitmessung mit `timeit` in einem Python-Skript

Mit Hilfe der `setup` Option können Anweisungen vor der eigentlichen Messung ausgeführt werden. Mehrere Anweisungen werden durch Semikolons oder Newlines getrennt:

```
def test():
    """Stupid test function"""
    L = []
    for i in range(100):
        L.append(i)

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Lst. C.4: Laufzeitmessung mit `timeit`, Einbinden von eigenen Modulen

C.1.2. `time.clock()` und `time.time()`

Eine schnelle (aber ungenauere) Messung kann man mit dem `time` Modul durchführen:

```
import time
t1 = time.clock()
for i in range(10000)
                "-".join(str(n) for n in range(100))
t2 = time.clock()
print((t2 - t1)/10000, " s")
```

Lst. C.5: Laufzeitmessung mit `time.clock()`

Als Beispiel soll eine Funktion zur paarweisen Berechnung der euklidischen Distanz zwischen $M = 1000$ Punkten mit $N = 3$ Dimensionen verwendet werden (http://jakevdp.github.io/blog/2013/06/15/numba-vs-cython-take-2/), die ein 1000 x 1000 Array zurück gibt.

Zunächst implementiert in reinem Python:

```
import numpy as np
X = np.random.random((1000, 3))
# X.shape = (1000, 3)

def pairwise_python(X):
    M = X.shape[0] # number of data points
    N = X.shape[1] # number of dimensions
    D = np.empty((M, M), dtype=np.float) # initialize result array
    for i in range(M):
        for j in range(M):
            d = 0.0
            for k in range(N):
                tmp = X[i, k] - X[j, k]
                d += tmp * tmp
```

```
15              D[i, j] = np.sqrt(d) # distance between X[i] and X[j]
16      return D
17  %timeit pairwise_python(X) # IPython helper function for speed calculations
```

Lst. C.6: Distanzberechnung mit Python

Im Test benötigt diese Implementierung 13.4s Rechenzeit, allerdings ist der Speicherbedarf minimal: Außer dem Ergebnisarray wird nur die Variable `tmp` benötigt.

C.2. Vektorisierung

Gerade bei wissenschaftlichen Problemstellungen sollte man zunächst versuchen, den Algorithmus in Vektor- bzw. Matrixschreibweise umformulieren: Nur dann können NumPys hochoptimierte Libraries für lineare Algebra ihre Stärken ausspielen. Numpy basiert auf ATLAS[1] (Automatically Tuned Linear Algebra Software), Intels MKL (Math Kernel Library) oder AMDs ACML (AMD Core Math Lib). Matlab™, Octave, Maple, Mathematica, R, Scilab und andere wissenschaftliche Sprachen binden die gleichen Libraries ein, bei entsprechend formulierten Problemstellungen gibt es daher auch keine dramatischen Performance-Unterschiede.

Durch Vektorisierung des Abstandsproblems erhält man folgenden zunächst schwer verständlichen Code:

```
1  ...
2  def pairwise_numpy(X):
3      return np.sqrt(((X[:, None, :] - X) ** 2).sum(-1))
4  %timeit pairwise_numpy(X)
```

Lst. C.7: Distanzberechnung, beschleunigt durch NumPy

`X[:, None, :]` indiziert ? , davon wird X subtrahiert. `sum(-1)` ist das Gleiche wie `sum(axis = -1)` und summiert über alle Elemente der letzen „Achse“ (Dimension). Das entspricht der `d += tmp * tmp` aus obigem Beispiel.

Im Test benötigt diese Implementierung 311 ms, ist also um einen Faktor 100 schneller, dafür wird ein temporäres Array mit 1000 x 1000 x 3 Elementen angelegt.

C.3. Just-In Time Compiler

Just-In-Time Compiler beschleunigen die Ausführung bereits während der Ausführung, Matlab hat JIT Support bereits seit einigen Jahren.

[1] eine an die jeweilige Hardware angepasste Implementierung der Libraries BLAS (Basic Linear Algebra Subprograms) und LAPack (Linear Algebra PACKage)

C.3.1. PyPy

Der bekannteste JIT Compiler für Python ist PyPy, quasi eine Parallelentwicklung zum „normalen“, in C implementierten Python Interpreter (CPython, nicht zu verwechseln mit Cython und ctypes). Nachteile sind:

Numpy ist nur teilweise in PyPy implementiert, daher werden auch scipy, matplotlib etc. nicht vernünftig unterstützt. PyPy ist daher für wissenschaftliche Probleme nur schlecht geeignet und eher *langsamer* als CPython mit NumPy!

Stabilität von CPython ist deutlich höher aufgrund des viel größeren Entwicklungsteams.

Kurz laufende Skripte werden aufgrund des Overheads für die Kompilierung nicht beschleunigt, eher im Gegenteil.

Support für Python 3.x ist bislang eher experimentell.

C.3.2. Numba

Eine interessante Entwicklung für wissenschaftliche Anwendungen ist **Numba** (http://numba.pydata.org/), das Python - Code mit minimalem Aufwand beschleunigen kann. Numba wird von Continuum Analytics entwickelt und lässt sich daher am einfachsten zusammen mit der Anaconda Distribution einsetzen. NumbaPro kann auch die GPU einbinden, ist aber nur mit kommerzieller oder akademischer Lizenz erhältlich.

```
1 from numba import double
2 from numba.decorators import jit, autojit
3
4 pairwise_numba = autojit(pairwise_python)
5
6 %timeit pairwise_numba(X)
```

Lst. C.8: Distanzberechnung, beschleunigt mit numba

In einem Benchmark „Numba vs. Cython: Take 2“ (http://nbviewer.ipython.org/url/jakevdp.github.io/downloads/notebooks/NumbaCython.ipynb) erreicht numba-optimierter Code die gleiche Geschwindigkeit wie Cython (s.u.) optimierter Code und das mit minimalem Aufwand.

C.4. Cython

Cython ist eine einfache Möglichkeit, den übrigen Code zu beschleunigen: Python Code wird zunächst automatisch in C-Code umgesetzt und dann in ein Erweiterungsmodul kompiliert, das später zur (Python-)Laufzeit dynamisch gelinkt wird. Ohne eine Zeile Code zu verändern, wird die Geschwindigkeit um ca. 30% gesteigert (siehe z.B. „Python for scientific computing: Where to start“ http://sjbyrnes.com/?page_id=67). Ein einfaches Beispiel von http://docs.cython.org: Der zu beschleunigende Code wird in File `*.pyx` geschrieben.

```
def say_hello_to(name):
    print("Hello %s!" % name)
```

Lst. C.9: Zu kompilierender Code `hello.pyx`

Außerdem benötigt man ein `setup.py` Skript mit Kompilierungsanweisungen:

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
  name = 'Hello world app',
  ext_modules = cythonize("hello.pyx"),
)
```

Lst. C.10: Setup-Skript `setup.py`

Die Kompilierung startet man mit `python setup.py build_ext --inplace`, die Option `--inplace` sorgt dafür, dass das kompilierte Modul aus dem gleichen Verzeichnis aufgerufen werden kann. Die kompilierte Erweiterung (`hello.so` unter *nix bzw. `hello.pyd` unter Windows) wird zur (Python-)Laufzeit dynamisch eingebunden mit:

```
from hello import say_hello_to
say_hello_to('Bob')
```

Lst. C.11: Python-Skript `name_calling.py`

Deutlich schneller (Faktor 5 ... 100) wird man mit Type-Deklarationen und ggf. Abschalten von Boundary Checks etc:

```
%%cython # IPython Magic, starting the compilation

import numpy as np
cimport cython
from libc.math cimport sqrt

@cython.boundscheck(False) # turn off boundary checks
@cython.wraparound(False)       # turn off Wrapping around
def pairwise_cython(double[:, ::1] X):
    cdef int M = X.shape[0]     # type declarations
    cdef int N = X.shape[1]     #   have the most effect
    cdef double tmp, d
    cdef double[:, ::1] D = np.empty((M, M), dtype=np.float64)
    for i in range(M):
        for j in range(M):
            d = 0.0
            for k in range(N):
                tmp = X[i, k] - X[j, k]
                d += tmp * tmp
            D[i, j] = sqrt(d)
    return np.asarray(D)
%timeit pairwise_cython(X)
```

Lst. C.12: Distanzberechnung, beschleunigt mit Cython

Hiermit wurde eine Laufzeit von 9.9 ms erreicht, langsamer als die numba-Optimierung!

Mit Cython kann man außerdem bestehende C-Libraries einbinden und kompilieren:

```
cdef extern from "math.h": double sin( double ) # access external C library
cdef double sinsq( double x): return sin(x*x)   # define C function
```

Lst. C.13: Verwenden von ext. C-Libraries

C.5. ctypes

Man kann mit dem **ctypes** Modul (Standard Python Library) aus Python heraus leicht Funktionen aus fremden DLLs / shared objects laden und aufrufen, hier gezeigt am Beispiel der Ansteuerung des USB-Messmoduls „Analog Discovery“ von Digilent / Analog Devices. Mit den folgenden Zeilen wird die `dwf.dll` importiert (aus „Create custom PC applications for Analog Discovery“, http://ez.analog.com/community/university-program/blog/2013/08/09/create-custom-pc-applications-for-analog-discovery

```
# load Digilent Waveforms DLL functions
from ctypes import *
cdll.LoadLibrary("dwf")  # looks for .dll in C:\Windows\System32
libdwf = CDLL("dwf")
```

In den meisten Fällen kann man den Funktionen gewöhnliche Python Variablen übergeben, aber manchmal müssen die Variablen als Referenz übergeben werden. In diesem Fall muss man sie als C - Variablen definieren wie hier das Handle für das Hardware Interface:

```
# interface handle
hdwf = c_int()
```

Um jetzt das erste gefundenen Analog Discovery Device zu öffnen, wird die open Funktion aufgerufen und hdwf als Referenz übergeben:

```
# open automatically the first available device
libdwf.FDwfDeviceOpen(-1, byref(hdwf))
```

Die Variable hdwf enthält jetzt den Handle mit dem das Hardware Interface in den folgenden Funktionsaufrufen adressiert wird. Beispiel: Schalte die positive 5 Volt Versorgung ein:

```
# enable positive supply
libdwf.FDwfAnalogIOChannelNodeSet(hdwf, 0, 0, 1)
```

Zum Schluss wird die Verbindung zum Device geschlossen:

```
# close all opened devices by this process
libdwf.FDwfDeviceCloseAll()
```

C.6. CFFI

C Foreign Function Interface for Python. Interact with almost any C code from Python, based on C-like declarations that you can often copy-paste from header files or documentation (http://cffi.readthedocs.io/). CFFI ist kein Bestandteil von Standardpython, kann aber z.B. in Anaconda leicht nachinstalliert werden (`conda install -c anaconda cffi=1.6.0`) http://eli.thegreenplace.net/2013/03/09/python-ffi-with-ctypes-and-cffi/

C.7. Weave und SWIG

Weave ist ein Untermodul von Scipy, mit dem man C- oder C++ - Code einbinden kann. **SWIG** ist eine andere Alternative, die z.B. im GnuRadio Projekt eingesetzt wird, einer flexibel programmierbaren Realtime DSP Umgebung: Hier verbindet Python geschwindigkeitsoptimierte C++ Module zur Signalverarbeitung flexibel miteinander, auch Scheduler und Testsignalerzeugung wurden in Python realisiert.

C.8. GPUs

GPUs werden mit pyCUDA, gnumpy, pyOpenCL zum Schwitzen gebracht, diese Module setzen auf Nvidias CUDA oder die offene OpenCL Bibliothek auf.

D. GUIs und Applications

D.1. Überblick

D.2. GUI aus Matplotlib-Widgets

```
axFSB = plt.axes([0.1, 0.15, 0.8, 0.03], axisbg='lightyellow')
sFSB  = Slider(axFSB, 'F_SB', 0.0, 0.5, valinit = F_SB0)

axBtReset = plt.axes([0.8, 0.025, 0.1, 0.04])
btReset = Button(axBtReset, 'Reset', color='lightyellow', hovercolor='red')

axBtQuit = plt.axes([0.6, 0.025, 0.1, 0.04])
btQuit = Button(axBtQuit, 'Quit', color='lightyellow', hovercolor='red')
#---------------------------------------------------------------------------
# Event loop starts here
#---------------------------------------------------------------------------
def update(event): # slider value is passed but not used here
    F_DB = sFDB.val # pass slider values
    F_SB = sFSB.val #   to F_DB and F_SB
    [bb, aa] = sig.iirdesign(F_DB*2, F_SB*2, A_DB, A_SB, ftype='ellip')
    [W, H]   = sig.freqz(bb, aa, 2048)
    h_lin.set_ydata(20*np.log10(abs(H)))  # recalculate + update data
    plt.draw()                            # update screen

def reset(event): sFDB.reset(); sFSB.reset()

def exitfunc(event): sys.exit(1)
#===========================================================================
# Call Event loop functions when one of the widgets is changed (= mouse event)
sFDB.on_changed(update)
sFSB.on_changed(update)

btReset.on_clicked(reset)
btQuit.on_clicked(exitfunc)

plt.show() # plot and wait
```

Lst. D.1: Filterdesign Applikation mit Matplotlib Widgets

Vorteile sind:

Wenige Abhängigkeiten:

Unabhängigkeit von Toolkit und Betriebssystem: Es werden nur Elemente der Matplotlib benötigt.

Nachteile sind:

Wenige Widgets: Eigentliche GUI-Elemente sind nur Button, Slider und Radiobutton. Es fehlen z.B. Dropdown-Listen und Text- oder numerische Eingabefelder

Schlichte Funktionalität: Der Event-Loop Mechanismus ist ziemlich einfach gestrickt

Kein grafisches Entwurfstool: Widgets müssen von Hand platziert und ausgerichtet werden

D.3. Qt-Widgets

D.3.1. Literatur & Videos

, „Learn Python GUI programming using Qt framework“ (https://www.udemy.com/python-gui-programming/)
Jason Fruit , „Python PySide/PyQt Tutorial“ (http://www.pythoncentral.io/series/

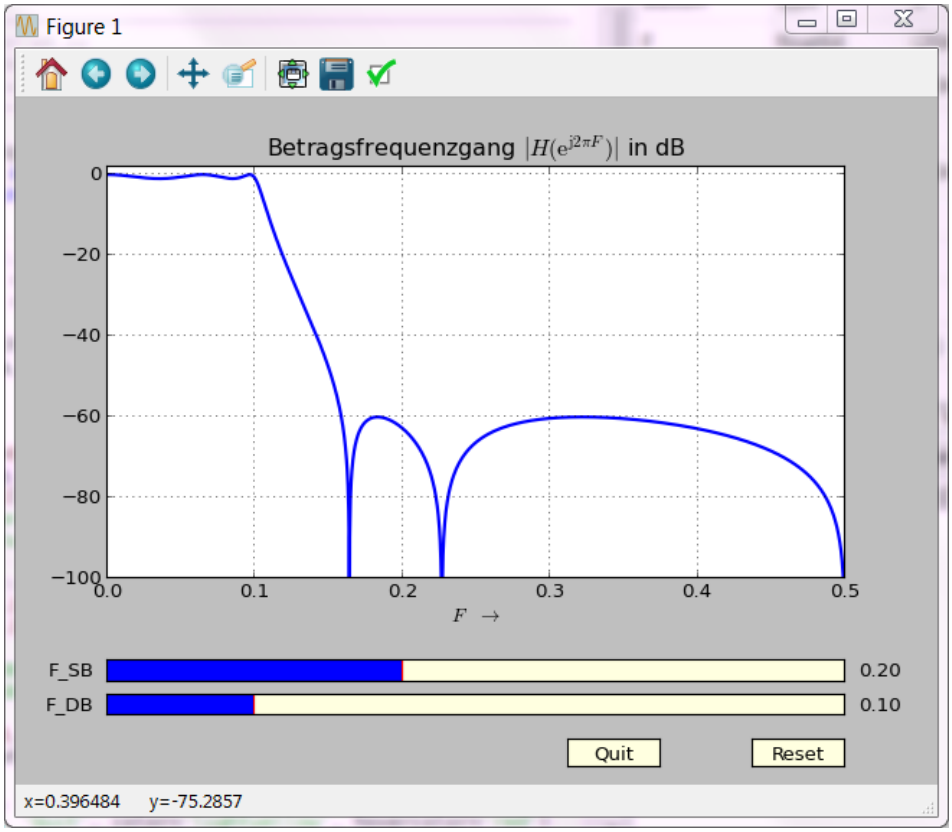


Abb. D.1.: GUI mit Matplotlib Widgets

python-pyside-pyqt-tutorial/). Ein Tutorial in acht Teilen für Qt unter Python (PySide and PyQt) mit ausführlichen Erklärungen, vielen Code-Beispielen und Referenzen. Das Tutorial startet mit einem „Hello World“ Widget und geht bis zum Einsatz von QWebView für die Darstellung von HTML-Code und Webseiten.

Jan Bodnar, „PyQt4 tutorial“ (http://zetcode.com/gui/pyqt4/)

Qt
PyQt
PySide

D.3.2. Signal-Slot-Mechanismus

[PyQt 4.9.6 Reference Guide]: One of the key features of Qt is its use of *signals* and *slots* to communicate between objects. Their use encourages the development of reusable components.

A signal is emitted when something of potential interest happens. A slot is a Python callable. If a signal is connected to a slot then the slot is called when the signal is emitted. If a signal isn't connected then nothing happens. The code (or component) that emits the signal does not know or care if the signal is being used.

The signal/slot mechanism has the following features:

- A signal may be connected to many slots.
- A signal may also be connected to another signal.
- Signal arguments may be any Python type.
- A slot may be connected to many signals.
- Connections may be direct (ie. synchronous) or queued (ie. asynchronous).
- Connections may be made across threads.
- Signals may be disconnected.

Beim alten Signal-Slot-Connect Mechanismus (nur PyQt < 5) muss die Signal-Signatur im C++ - Stil m*exakt* aufgeschrieben werden, ansonsten wird kein Signal abgefeuert und es wird auch keine Fehlermeldung produziert:

```
self.connect(the_button, SIGNAL('clicked()'), self.on_hello)
```

Dieser Mechanismus wird ab PyQt 5 nicht mehr unterstützt, also auch nicht mehr SIGNAL() oder SLOT())!

Für neue Projekte sollte daher ausschließlich der neue Connect Mechanismus (PyQt > 4.4) verwendet werden, der einfacher und sicherer (Fehlermeldung) ist und mit einer sauberen Objektorientierung „pythonesker“ ist:

```
the_button.clicked.connect(self.on_hello)
```

```
A_SB = 60   # min. Sperrdämpfung im Stoppband in dB

class AppForm(QMainWindow):

    def __init__(self, parent=None):

        QMainWindow.__init__(self, parent)
        self.setWindowTitle('Demo: IIR-Filter Design using PyQt with Matplotlib')

        self.create_main_frame()

        self.txtbox_F_DB.setText('0.1')
        self.txtbox_F_SB.setText('0.2')

        self.on_draw()

    def on_draw(self):
        """ Redraws the figure
        """
        self.F_SB = float(unicode(self.txtbox_F_SB.text()))
        self.F_DB = float(unicode(self.txtbox_F_DB.text()))
        self.FiltType = unicode(self.combo_FiltType.currentText())

        [bb,aa] = sig.iirdesign(self.F_DB*2, self.F_SB*2, A_DB, A_SB, ftype=self.FiltType
            )
        [W,H] = sig.freqz(bb,aa,N_FFT) # calculate H(W) for W = 0 ... pi
        F = W / (2 * np.pi)

        # clear the axes and redraw the plot
```

```
29          #
30          self.axes.clear()
31          self.axes.grid(self.cb_grid.isChecked())
32          self.axes.axis([0, 0.5, -100, 2])
33
34          self.axes.plot(F,20*np.log10(abs(H)),
35                         lw = self.slider.value())
36          self.axes.set_xlabel(r'$f \; \rightarrow$')
37          self.axes.set_ylabel(r'$|H(\mathrm{e}^{\mathrm{j} \Omega})|\; \rightarrow $')
38          self.axes.set_title(r'Betragsfrequenzgang')
39
40          self.canvas.draw()
41
42      def create_main_frame(self):
43          self.main_frame = QWidget()
44
45          # Create the mpl Figure and FigCanvas objects.
46          # 5x4 inches, 100 dots-per-inch
47          #
48          self.dpi = 100
49          self.fig = Figure((5.0, 4.0), dpi=self.dpi)
50          self.canvas = FigureCanvas(self.fig)
51          self.canvas.setParent(self.main_frame)
52
53          self.axes = self.fig.add_subplot(111)
54
55          # Create the navigation toolbar, tied to the canvas
56          #
57          self.mpl_toolbar = NavigationToolbar(self.canvas, self.main_frame)
58
59          # Other GUI controls: SIGNAL definitions and connections to SLOTS
60          #
61          self.txtbox_F_SB = QLineEdit()
62          self.txtbox_F_SB.setFixedWidth(100)
63          self.connect(self.txtbox_F_SB, SIGNAL('editingFinished ()'), self.on_draw)
64          lbl_F_SB = QLabel('F_SB:')
65
66          self.txtbox_F_DB = QLineEdit()
67          self.txtbox_F_DB.setFixedWidth(100)
68          self.connect(self.txtbox_F_DB, SIGNAL('editingFinished ()'), self.on_draw)
69          lbl_F_DB = QLabel('F_DB:')
70
71          self.btn_draw = QPushButton("&Draw")
72          self.connect(self.btn_draw, SIGNAL('clicked()'), self.on_draw)
73
74          self.btn_quit = QPushButton("&Quit")
75          self.connect(self.btn_quit, SIGNAL('clicked()'), self, SLOT("close()"))
76
77          self.cb_grid = QCheckBox("Show &Grid")
78          self.cb_grid.setChecked(True)
79          self.connect(self.cb_grid, SIGNAL('stateChanged(int)'), self.on_draw)
80
81          self.combo_FiltType = QComboBox()
82          FiltType = ['butter','cheby1','cheby2','ellip']
83          for i in FiltType: self.combo_FiltType.addItem(i)
84          lbl_FiltType = QLabel('Filtertyp')
85          self.combo_FiltType.activated[str].connect(self.on_draw)
86
87          lbl_lw = QLabel('Line width:')
88          self.slider = QSlider(Qt.Horizontal)
89          self.slider.setRange(1, 5)
90          self.slider.setValue(2)
91          self.slider.setTracking(True)
92          self.slider.setTickPosition(QSlider.NoTicks)
93          self.connect(self.slider, SIGNAL('valueChanged(int)'), self.on_draw)
```

```
94
95          #============================================
96          # Layout with box sizers
97          #============================================
98
99          hbox1 = QHBoxLayout()
100         for w in [ lbl_F_DB, self.txtbox_F_DB,  lbl_FiltType,
101                    self.combo_FiltType, self.btn_draw, self.btn_quit]:
102             hbox1.addWidget(w)
103             hbox1.setAlignment(w, Qt.AlignVCenter)
104
105         hbox2 = QHBoxLayout()
106         for w in [ lbl_F_SB, self.txtbox_F_SB, self.cb_grid,
107                     lbl_lw, self.slider]:
108             hbox2.addWidget(w)
109             hbox2.setAlignment(w, Qt.AlignVCenter)
110
111         vbox = QVBoxLayout()
112         vbox.addWidget(self.mpl_toolbar)
113         vbox.addWidget(self.canvas)
114         vbox.addLayout(hbox1)
115         vbox.addLayout(hbox2)
116
117         self.main_frame.setLayout(vbox)
118         self.setCentralWidget(self.main_frame)
119
120 def main():
121     app = QApplication(sys.argv)
122     form = AppForm()
123     form.show()
124     app.exec_()
125
126 if __name__ == "__main__":
127     main()
```

Lst. D.2: Filterdesign Applikation mit Qt Widgets und Matplotlib Canvas

D.3.3. Textbasiertes Layout

Die simpleste (unkomfortabelste) Layoutmethode ist es, *absolute Koordinaten* zu verwenden, z.B.

```
1 cb.setGeometry(10, 10, 150, 20)
```

GridLayout ist eine andere Art Widgets zu platzieren:

```
1 grid = QGridLayout()
2 grid.addWidget(dateLabel, 0, 0)
3 grid.addWidget(self.fromComboBox, 1, 0)
4 grid.addWidget(self.fromSpinBox, 1, 1)
5 grid.addWidget(self.toComboBox, 2, 0)
6 grid.addWidget(self.toLabel, 2, 1)
7 self.setLayout(grid)
```

Lst. D.3: GridLayout für Qt Widgets

Eine weitere komfortable Methode ist das *Box-Modell*.

D.3.4. Layout mit Qt-Designer

Am komfortabelsten ist es den Qt-Designer zu verwenden, mit dem die gesamte GUI grafisch „zusammengeklickt“ werden kann (siehe z.B. http://pyqt.sourceforge.net/Docs/PyQt4/designer.html). Das GUI wird in einem File `tollesGUI.ui` abgespeichert, das mit dem Skript `pyuic4.bat -x tollesGUI.ui -o tollesGUI.py` zu einem Pythonfile umgewandelt wird.

Alternativ kann mit `from PyQt4 import uic; uic.loadUi('tollesGUI.ui', self)` der *.ui File direkt eingebunden werden (siehe Listing D.4 aus #http://stackoverflow.com/questions/2398800/linking-a-qtdesigner-ui-file-to-python-pyqt.

```
1  import sys
2  from PyQt4 import QtGui, uic
3
4  class MyWindow(QtGui.QMainWindow):
5      def __init__(self):
6          super(MyWindow, self).__init__()
7          uic.loadUi('tollesGUI.ui', self)
8          self.show()
9
10 if __name__ == '__main__':
11     app = QtGui.QApplication(sys.argv)
12     window = MyWindow()
13     sys.exit(app.exec_())
```

Lst. D.4: Einbinden eines Qt-Designer *.ui - Files in ein Pythonskript

D.3.5. API #1, API #2 und PySide

Seit PyQt4.6 gibt es zwei APIs (Application Program Interface), API #1 (das Original) und API #2 (neu). API #2 ist leichter zu benutzen, „pythonischer“, dort wurden allerdings `QString` und `QVariant` durch Python-Objekte ersetzt (`str` und `??`.

API #1 ist der Default für PyQt4.x mit Python 2.x und bis PyQt4.5 mit Python 3.x. API #2 ist der Default für PyQt4.6+ mit Python 3.x, und wird auch von PySide genutzt. Ab PyQt5 kann nur noch die API #2 verwendet werden.

Es ist vor allem wichtig zu wissen, **dass** es zwei API Varianten gibt, da gelegentlich entsprechende Fehlermeldungen auftreten.

D.3.6. Matplotlib in PyQt

http://matplotlib.org/examples/user_interfaces/embedding_in_qt4.html

D.3.7. Realtime-Applikationen mit PyQwt

PyQwt provides Python bindings to the C++ Qwt library, which is a set of classes extending the Qt framework with plotting and other widgets for scientific and engineering applications (dials, knobs, sliders). Qwt ist eine Unterbibliothek von Qt, die Widgets können daher problemlos gemeinsam mit Qt Widgets verwendet und mit Qt Designer platziert werden.

PyQwt ist deutlich „leichter“ und schneller als die Matplotlib (Binaries sind deutlich kleiner), und ist daher für RealTime Plotting deutlich besser geeignet. Es unterstützt allerdings auch weniger Plot Optionen, auch die Qualität der Plots ist nicht ganz so hoch (z.B. kein Latex).

Die Dokumentation ist deutlich schlechter als die der Matplotlib, am besten hält man sich hier an Codebeispiele.

Siehe z.B. „Realtime FFT Audio Visualization with Python“ (http://www.swharden.com/blog/2013-05-09-realtime-fft-audio-visualization-with-python/) und „Realtime Data Plotting in Python“ (http://www.swharden.com/blog/2013-05-08-realtime-data-plotting-in-python/).

Vergleich:
http://eli.thegreenplace.net/2009/05/23/more-pyqt-plotting-demos/
http://eli.thegreenplace.net/2009/06/05/plotting-in-python-matplotlib-vs-pyqwt/

D.4. Kivy

> „Kivy - Open source Python library for rapid development of applications that make use of innovative user interfaces, such as multi-touch apps.“

Kivy (http://kivy.org) ist eine sehr interessante Alternative zu Qt wenn es um die Entwicklung von Applikationen auch für Tablets oder Smartphones geht. Die Graphics Engine nutzt Hardwarebeschleunigung und ist daher auch für Devices mit schwacher Rechenleistung (Smartphones, Raspberry Pi) geeignet.

E. Source Code

E.1. Tipps und Tricks zur Matplotlib

E.1.1. Subplots mit gemeinsamem Zoom

```
ax1 = plt.subplot(2,1,1)
ax1.plot(...)
ax2 = plt.subplot(2,1,2, sharex=ax1)
ax2.plot(...)
```

Lst. E.1: Gemeinsamer Zoom von Subplots

E.1.2. Plots mit doppelter y-Achse (twinx) und Legende

Man kann innerhalb eines Subplot-Fenster zwei Plots mit unterschiedlicher Skalierung und Beschriftung der y-Achse erzeugen, indem man ein zweites Axes-Objekt `ax2` von einem bereits definierten `ax1` und der Methode `twinx()` ableitet.

Problematisch ist es dann eine Legende zu erhalten, die beide Plots enthält, da für jedes Axes-Objekt eine eigene Legende erzeugt wird. Dies kann man umgehen, indem man die geplotteten Objekte und Labels „einsammelt“ und in einer Legende wiedergibt.

```
ax1 = fig.add_subplot(111)
ax1.plot(time, values, label = 'Values 1')
ax2 = ax1.twinx() # second y-axis with separate scaling
ax2.plot(time, other_values, label = 'Values 2')

# ask matplotlib for the plotted objects and their labels
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax1.legend(lines + lines2, labels + labels2)
```

Lst. E.2: Plots mit doppelter y-Achse

E.1.3. Formatierung der Beschriftung mit rcParams

Alternativ können die gleichen Parameter in der matplotlibrc als Default-Einstellungen eingetragen werden.

```
params =  {'axes.labelsize': 18,
           'text.fontsize': 18,
           'legend.fontsize': 14,
           'xtick.labelsize': 14,
           'ytick.labelsize': 14,
           'figure.figsize': [8.0 , 6.0]}
plt.rcParams.update(params)
```

Lst. E.3: Fonts und Beschriftung ändern

E.1.4. Formatierung der Achsen-Lables

```
formatter = plt.ScalarFormatter(useMathText = True) # nicht als Attribut ???
formatter.set_scientific(True) # Achenbeschriftung in scientific notation
formatter.set_powerlimits((-3,3)) # ... wenn x < 1e-3 oder x > 1e3
formatter.set_useMathText(True)  # hübsche Formatierung für Multiplikator

figure(1)
ax1 = subplot(1,1,1)
stem(t2,y2)
ax1.xaxis.set_major_formatter(formatter)
#plt.gca().xaxis.set_major_formatter(formatter) # alternativ
```

Lst. E.4: Formatierung der Achsen ändern

E.2. Source-Code zu Bildern im Text

Hier wird der Source-Code für einige aufwändigere Bilder gezeigt:

```
# (c) Rougier NP, Droettboom M, Bourne PE (2014) Ten Simple Rules for Better Figures.
# PLoS Comput Biol 10(9): e1003833. doi:10.1371/journal.pcbi.1003833
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure(figsize=(8,4))
n = 256
X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C,S = np.cos(X), np.sin(X)
plt.plot(X,C), plt.plot(X,S)
savefig("figure-4-left.pdf")
show()
```

Lst. E.5: Plot mit Matplotlib-Defaults

```
# (c) Rougier NP, Droettboom M, Bourne PE (2014) Ten Simple Rules for Better Figures.
# PLoS Comput Biol 10(9): e1003833. doi:10.1371/journal.pcbi.1003833
from pylab import *
figure(figsize=(8,5), dpi=80)
subplot(111)
X = np.linspace(-np.pi, np.pi, 256,endpoint=True)
C,S = np.cos(X), np.sin(X)
plot(X, C, color="blue", linewidth=2.5, linestyle="-", label="cosine")
plot(X, S, color="red", linewidth=2.5, linestyle="-", label="sine")
```

```
ax = gca()
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
xlim(X.min()*1.1, X.max()*1.1)
xticks([-np.pi, -np.pi/2, 0, np.pi/2, np.pi],
[r'$-\pi$', r'$-\pi/2$', r'$0$', r'$+\pi/2$', r'$+\pi$'])
ylim(C.min()*1.1,C.max()*1.1)
yticks([-1, +1],
[r'$-1$', r'$+1$'])
legend(loc='upper left')
t = 2*np.pi/3
plot([t,t],[0,np.cos(t)],
color ='blue', linewidth=1.5, linestyle="--")
scatter([t,],[np.cos(t),], 50, color ='blue')
annotate(r'$sin(\frac{2\pi}{3})=\frac{\sqrt{3}}{2}$', xy=(t, np.sin(t)), xycoords='data',
xytext=(+10, +30), textcoords='offset points', fontsize=16,
arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
plot([t,t],[0,np.sin(t)],
color ='red', linewidth=1.5, linestyle="--")
scatter([t,],[np.sin(t),], 50, color ='red')
annotate(r'$cos(\frac{2\pi}{3})=-\frac{1}{2}$', xy=(t, np.cos(t)), xycoords='data',
xytext=(-90, -50), textcoords='offset points', fontsize=16,
arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=.2"))
for label in ax.get_xticklabels() + ax.get_yticklabels():
label.set_fontsize(16)
label.set_bbox(dict(facecolor='white', edgecolor='None', alpha=0.65 ))
savefig("figure-4-right.pdf")
show()
```

Lst. E.6: Verbesserter Plot

```
# (c) Rougier NP, Droettboom M, Bourne PE (2014) Ten Simple Rules for Better Figures.
# PLoS Comput Biol 10(9): e1003833. doi:10.1371/journal.pcbi.1003833
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
plt.xkcd()
X = np.linspace(0,2.32,100)
Y = X*X - 5*np.exp(-5*(X-2)*(X-2))
fig = plt.figure(figsize=(12,5), dpi=72,facecolor="white")
axes = plt.subplot(111)
plt.plot(X,Y, color = 'k', linewidth=2, linestyle="-", zorder=+10)
axes.set_xlim(X.min(),X.max())
axes.set_ylim(1.01*Y.min(), 1.01*Y.max())
axes.spines['right'].set_color('none')
axes.spines['top'].set_color('none')
axes.xaxis.set_ticks_position('bottom')
axes.spines['bottom'].set_position(('data',0))
axes.yaxis.set_ticks_position('left')
axes.spines['left'].set_position(('data',X.min()))
axes.set_xticks([])
axes.set_yticks([])
axes.set_xlim( 1.05*X.min(), 1.10*X.max() )
axes.set_ylim( 1.15*Y.min(), 1.05*Y.max() )
t = [10,40,82,88,93,99]
plt.scatter( X[t], Y[t], s=50, zorder=+12, c='k')
plt.text(X[t[0]]-.1, Y[t[0]]+.1, "Industrial\nRobot", ha='left', va='bottom')
plt.text(X[t[1]]-.15, Y[t[1]]+.1, "Humanoid\nRobot", ha='left', va='bottom')
```

```
plt.text(X[t[2]]-.25, Y[t[2]], "Zombie", ha='left', va='center')
plt.text(X[t[3]]+.05, Y[t[3]], "Prosthetic\nHand", ha='left', va='center')
plt.text(X[t[4]]+.05, Y[t[4]], "Bunraku\nPuppet", ha='left', va='center')
plt.text(X[t[5]]+.05, Y[t[5]], "Human", ha='left', va='center')
plt.text(X[t[2]]-0.05, 1.5, "Uncanny\nValley", ha='center', va='center', fontsize=24)
plt.ylabel("- Comfort Level +",y=.5, fontsize=20)
plt.text(.05, -.1, "Human Likeness ->",ha='left', va='top', color='r', fontsize=20)
X = np.linspace(0,1.1*2.32,100)
axes.fill_between(X, 0, -10, color = '0.85', zorder=-1)
axes.fill_between(X, 0, +10, color = (1.0,1.0,0.9), zorder=-1)
#X = np.linspace(1.652,2.135,100)
X = np.linspace(1.5,2.25,100)
Y = X*X - 5*np.exp(-5*(X-2)*(X-2))
axes.fill_between(X, Y, +10, color = (1,1,1), zorder=-1)
axes.axvline(x=1.5,ymin=0,ymax=1, color='.5', ls='--')
axes.axvline(x=2.25,ymin=0,ymax=1, color='.5', ls='--')
plt.savefig("figure-8.pdf")
plt.show()
```

Lst. E.7: XKCD-Plot „Uncanny Valley“

Literaturverzeichnis

[PS97] W. Putnam and J. O. Smith, *Design of Fractional Delay Filters Using Convex Optimization*, IEEE ASSP Workshop on Applications of Signal Processing to Audio and Acoustics, Oct. 1997.

[Smi11] J. O. Smith, *Interpolated delay lines, ideal bandlimited interpolation, and fractional delay filter design*, Lecture Notes, February 2011, https://ccrma.stanford.edu/~jos/Interpolation/Interpolation.pdf.

[Smi14] _____, *Bandlimited Interpolation - Introduction and Algorithm (Digital Audio Resampling Home Page)*, 2014, https://ccrma.stanford.edu/~jos/resample/resample.html.