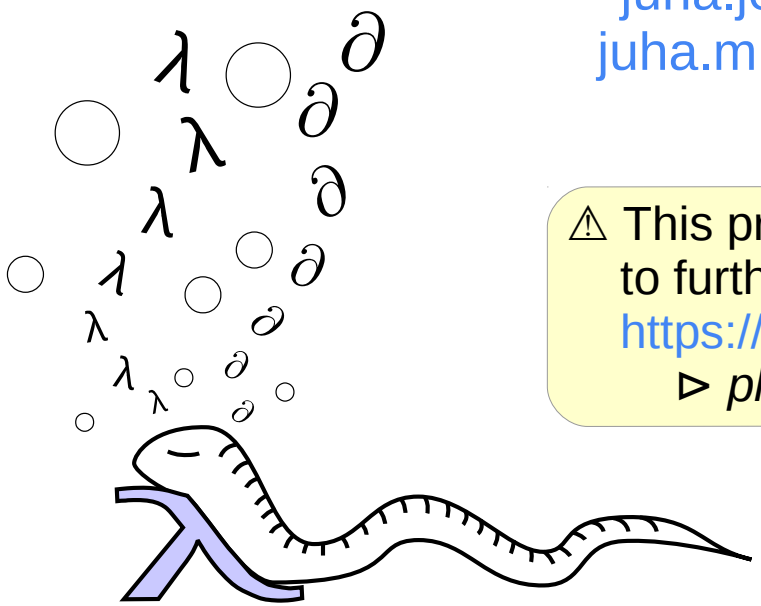
Programming *matters*

Experiences of Python in scientific computing

Juha Jeronen
juha.jeronen@jyu.fi ⇥ 05/2019
juha.m.jeronen@gmail.com → ∞



⚠ This presentation contains **many** weblinks to further reading. Get the slides at:
https://github.com/Technologicat/python-3-scicomp-intro
▷ *physsem2019* ▷ *physsem2019.pdf*

4.4.2019, Tampere University
FYS-1556 Physics Seminar 2018–19

UNIVERSITY OF JYVÄSKYLÄ

My background

$$\psi = \frac{1}{\sqrt{3}}|(\mathrm{IT}, \mathrm{CS})\rangle + \frac{1}{\sqrt{3}}|\mathrm{M}\rangle + \frac{1}{\sqrt{3}}|(\mathrm{P}, \mathrm{ES})\rangle$$

where

IT = information technology, CS = computer science

...with some software engineering, too

M = mathematics

P = physics, ES = engineering sciences

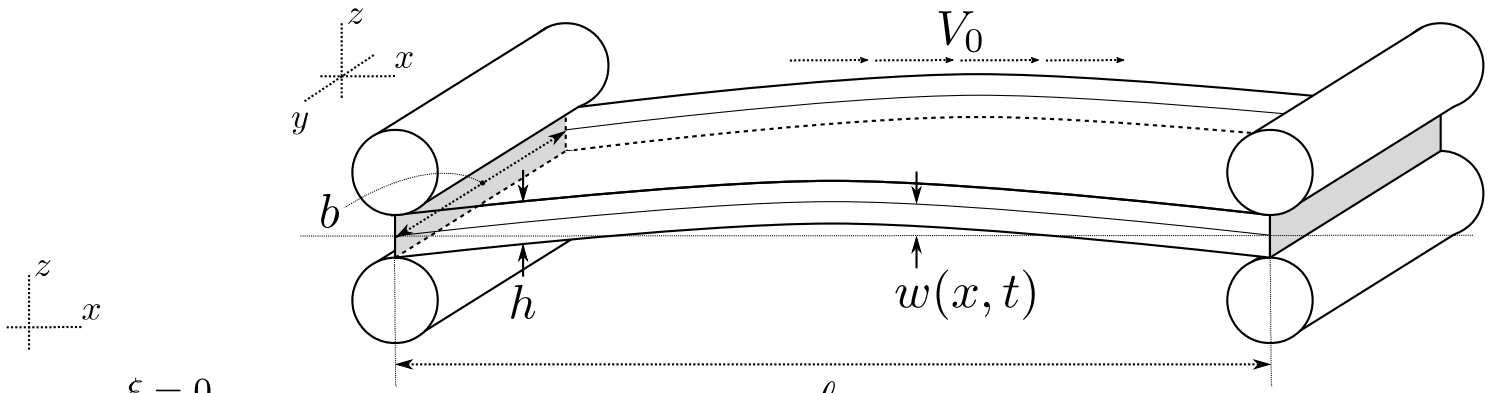
- Software engineer 1999–2007, Jutel Oy
- MSc 2008, University of Oulu, mathematics
- PhD 2011, University of Jyväskylä, information technology [thesis]
- Postdoc researcher 2012–, mostly at JYU; at TUT 2017–2018
  - Gave a Python course at TUT in 2018 [5 ECTS; material]

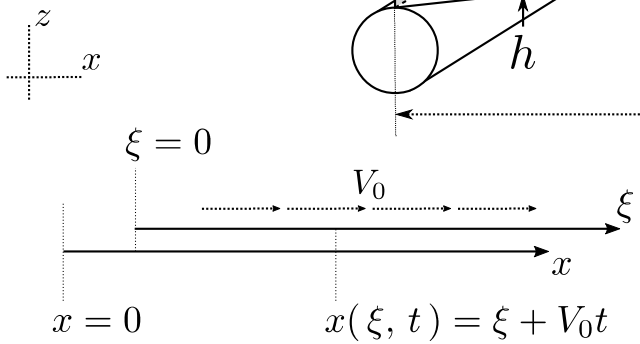
- Research topic: *axially moving materials* (+ some energy harvesting)
- Programming background before Python: C, C++, Perl, Java, MATLAB
- Python as primary programming language since 2012
  - ...with some experiments ([1] [2]) in Racket

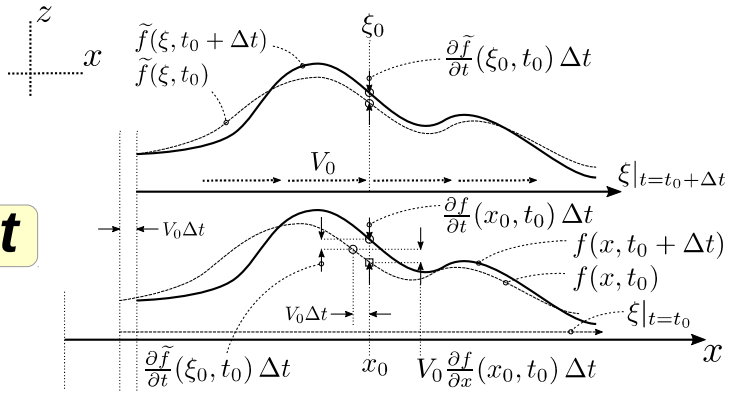
My GitHub.

Axially moving materials, in 3 minutes

$$m\frac{\partial^2 w}{\partial t^2} + 2mV_0\frac{\partial^2 w}{\partial x \partial t} + (mV_0^2 - T_0)\frac{\partial^2 w}{\partial x^2} + D\frac{\partial^4 w}{\partial x^4} = 0$$







- ***A solid, flowing through domain of interest***
- Applications e.g. in:
  - Process industry: **paper**, steel, textiles
  - Rotating storage media: hard disks, optical disc drives
  - Newspaper printing, band saw blades, tape drives, …
- My research topic: runnability of paper machines (stability analysis)
- Our books: *Mechanics of moving materials* (2014, SMIA vol. 207),
  Stability of axially moving materials (to appear 2019, SMIA)

“Programming matters”?

- Programming is ***codification of imperative knowledge*** (SICP 2e, Abelson & Sussman, 1996).
  - *Algorithms*: how to solve (particular) problems
  - *Data structures*: how to store data efficiently for different use cases
- Usually machine-readable (executable), but not central to the idea.
- Contrast pure mathematics, which is *declarative knowledge* – e.g. that something exists, but (often) no constructive proof to actually compute it.
- ***Paradigms*** (e.g. imperative, functional): *ways to structure that knowledge*.
- ***Languages***: *tools to think in*.

- **Programming languages are not equal**; variation in paradigms supported, focus/target audience, level of abstraction, type system, verbosity, feature set, …
  - *A language that doesn't affect the way you think about programming, is not worth knowing*. –*Alan Perlis, Epigrams on Programming* (1982)
  - *Haskell*: pure functional, focus on category theory, high level, concise, statically typed with parametric types (e.g. any “*a*” instead of int, double, …) and automatic type inference.
  - *Lisp* (family; see Racket for a modern Lisp): impure functional, language-oriented (extensible language defined partly by the programmer), high level, concise, dynamically typed. Has closures. The Scheme subfamily (including Racket) has continuations.
  - ***Python***: object-oriented, imperative, impure functional, focus on readability, high level, concise, dynamically typed, duck-typed. Has closures.
  - *C*, *Fortran*: imperative, procedural, respectively for systems programming and raw number crunching, low level, verbose, statically typed (in a very rudimentary, hardware-oriented way)
  - *C++*: imperative, object-oriented, low level, verbose, statically typed
  - *Prolog*: declarative, logic-oriented, high level (embedded into Racket as Racklog)

- **Practical implications**: human efficiency, maintainability, potential for automated analysis, …

“Programming matters”?

- ***Implementing numerical solvers is software engineering***
  - Computational research is at least 1/3 software engineering
    ...the other 2/3 divided between mathematics and writing papers
  - To be effective and efficient in such research, tools and practices developed in the software industry are extremely valuable

- **Problem**: *software is complex*
  - But humans can only work with a limited amount of complexity
  - Solution: reduce *visible* complexity, building a tower of abstractions
    - Efficiency? 80/20; also clock cycles cheap, human time expensive
    - So, push bits in Fortran, C, C++, Cython…, but write the 80% in an appropriate **high-level** language → wide-spectrum programming

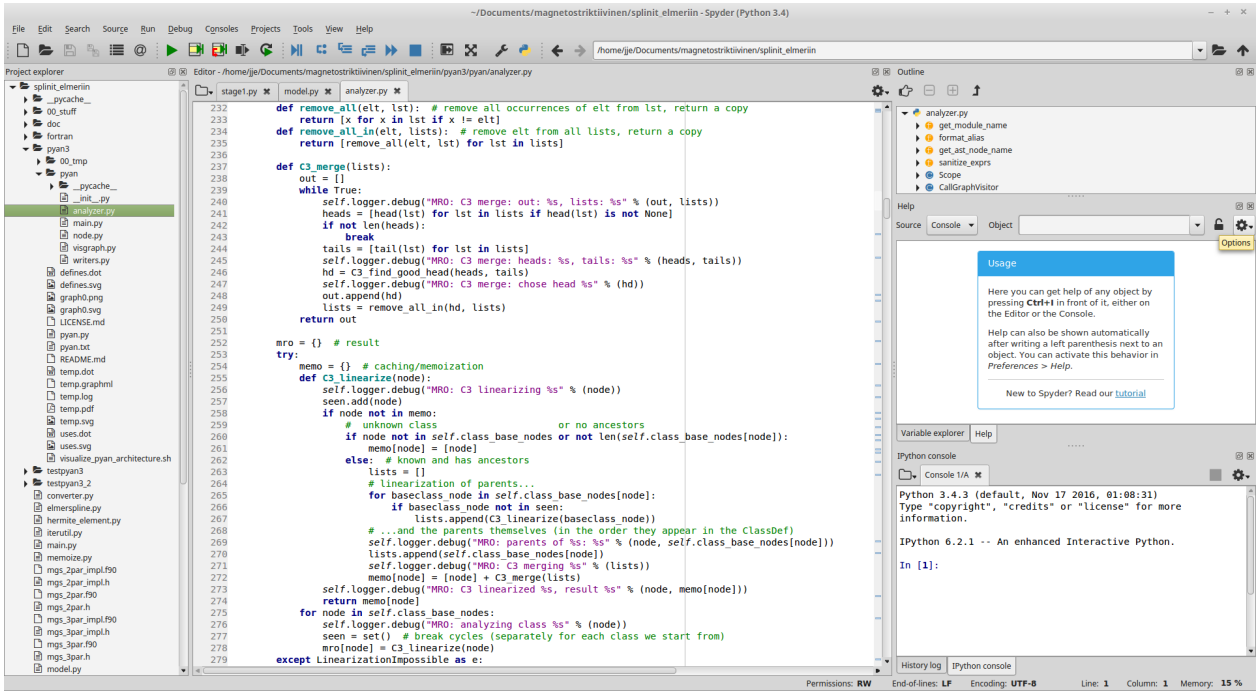
- A good language (for a given task) minimizes the impedance mismatch between the language and the problem domain ⇒ ***shorter programs***
  - Less work to write, easier to maintain, easier to spot errors
    ...and *much* easier to read and understand 6 months later
  - Similarly to how notation matters in mathematics (see also Leibniz)

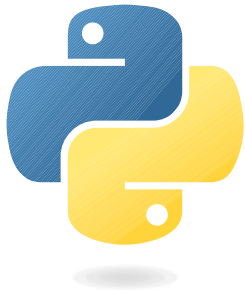
Python?

A fairly long history:

- Python 1.0: *1991*
- Python 2.0: *2000*
- Python 3.0: *2008*

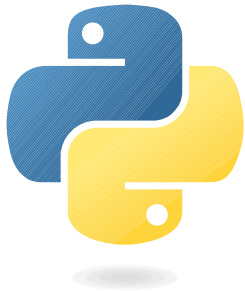
See a family tree of languages.





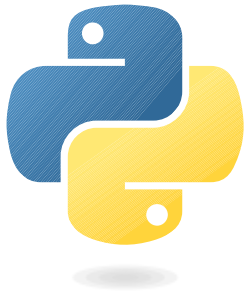
What & why for numerics

- Clear, general-purpose, high-level, well-designed complete programming solution
- Easy to learn
- Focus on clarity: often Python programs look clear, making it easier to return to old code later
- Open source; repeatability and transparency of science
- Free of cost; no need for licenses
- “Complete” includes numerics; a viable competitor for MATLAB
- Suitable for wide-spectrum programming, especially numerics



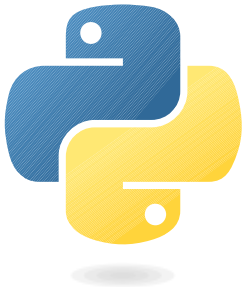
Why now?

- Python 3 is a sufficiently stable platform to build science on
- Rise and popularization of Python during the last 15 years
- Python now part of the mainstream of programming: many libraries, extensive help available on the internet (especially Stack Overflow)
- IEEE Spectrum 2017: Python the most popular language worldwide



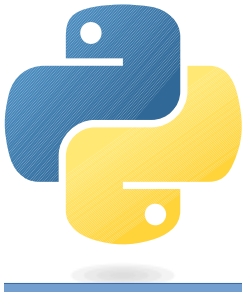
Where MATLAB is good

- Quality-of-life features of a commercial product
  - All-in-one
  - Attempts to do some things automatically for the user, e.g. just-in-time (JIT) compilation
  - Community focused solely on numerics
  - If an algorithm is not in MATLAB, it can likely be found in MATLAB File Exchange
- Polished integrated development environment (IDE) for scientists
- Interactive plot editor
- Some things easier to do than in Python (e.g. 3D plotting)
- SimuLink: graphical block model simulator



Where Python is good

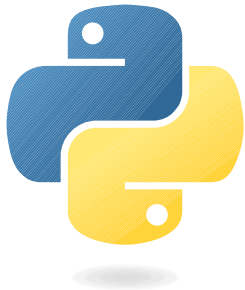
- Elegant language, in which it is easy to write clear code
- Control: does only what you explicitly tell it to
- Software ecosystem with dependency management (PyPI, `pip`)
  - A huge number of libraries *for anything a computer can do*
  - A sensibly sized numerics community, too
- Free of cost; no need for licenses
- Open; repeatability and transparency of science
  - In practice, also the libraries are open source. Sometimes a library already does 99% of what you need...



“Python”

- Technically speaking, “*Python*” is a specification, like “*C*” or “*Fortran*”
  - Several implementations: *CPython*, *Jython*, *IronPython*, *PyPy*
  - *CPython* however de facto standard; for most == Python

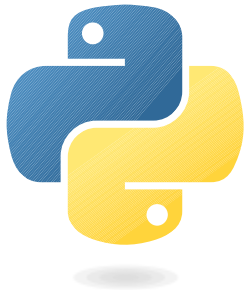
- Python 3 vs. Python 2
  - Python 3: current version (2019, v3.7.3)
    - Also unofficially known as *py3k*, *Python 3000*
  - Python 2: legacy (2010, v2.7, support ends by 2020)
    - Python 2.x ends at 2.7: *Python 2.8 Un-release Schedule*
  - Backwards incompatibilities; but devs promised not to do it again
  - Practically all projects have migrated to Python 3
  - This presentation: Python 3



Zen of Python, The

A guiding philosophy for the design of the Python language, as well as many Python programs. Consists of 20 aphorisms, 19 of which have been written down:

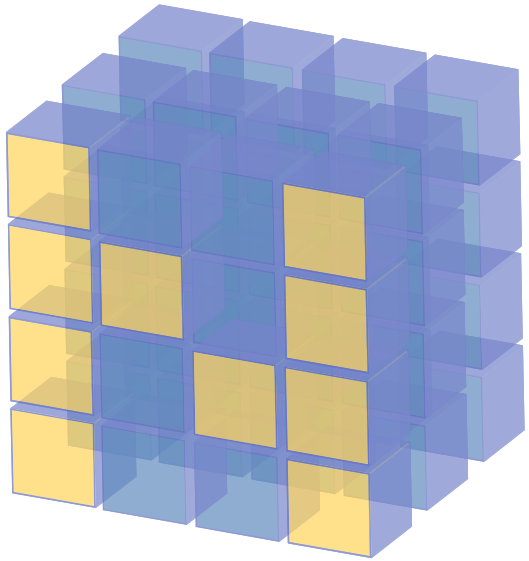
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than **right** now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!



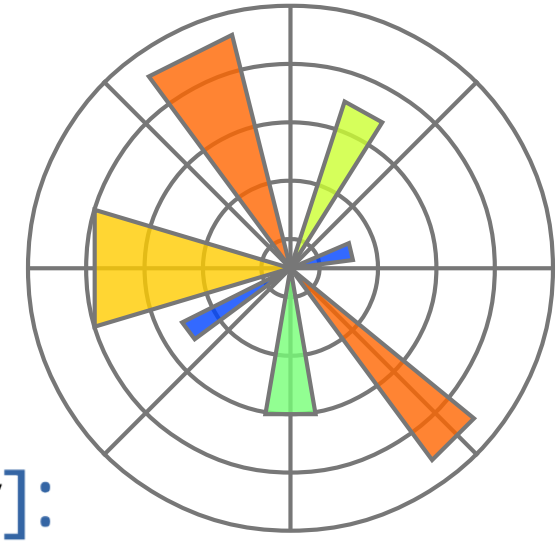
–Tim Peters

Scientific Python

Welcome to the bazaar



Cython



IP[y]:

ANACONDA

Numba

Tools

http://ipython.org/

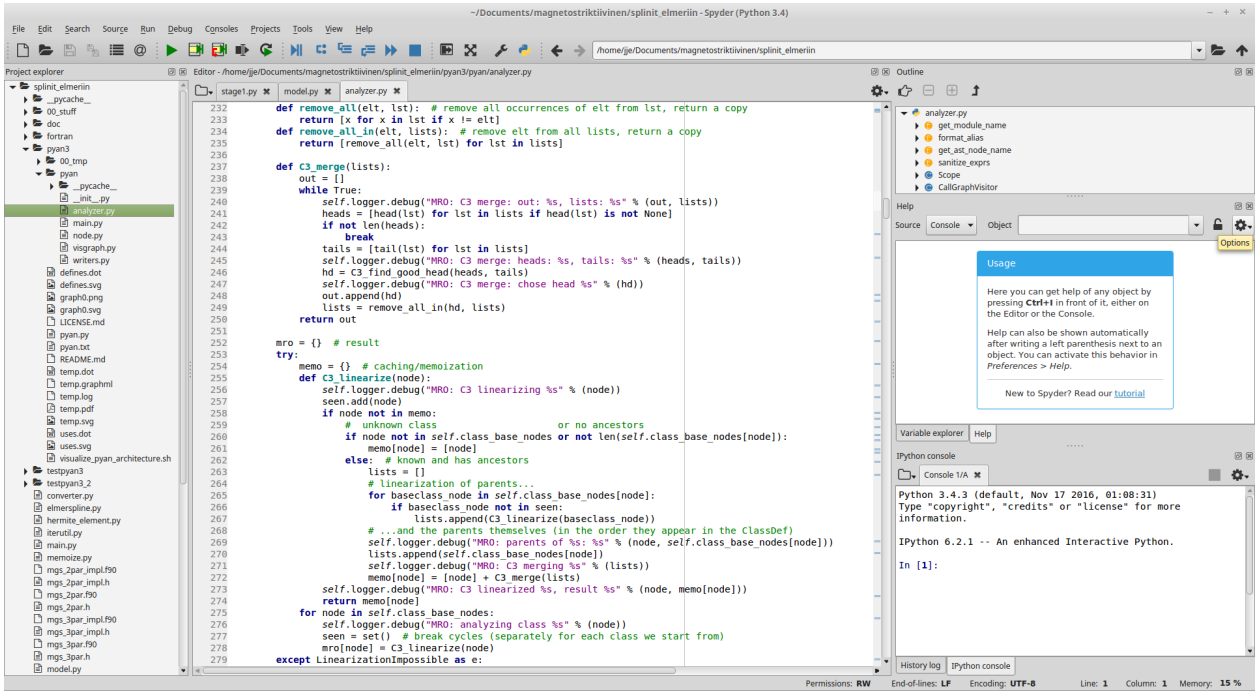
https://www.anaconda.com/

https://github.com/spyder-ide

- **IPython**: advanced command line
- **Jupyter**: IPython's graphical cousin, based on a Mathematica-style *notebook* approach
- **Spyder**: integrated development environment (IDE)
- **Anaconda**: scientific Python distribution
  - Perhaps the easiest way to install Python and its scientific libraries.

Spyder IDE

- The **S**cientific **PY**thon **D**evelopment **E**nvi**R**onment
- Preinstalled in Anaconda
- Designed for scientists
- MATLAB style IDE
- Matplotlib integration
- Debugger
- Profiler
- Static code analyzer
- REPL (IPython/Jupyter) read-eval-print-loop



https://github.com/spyder-ide

- Mostly well balanced between scientific use oriented, interactive, and software development oriented features.
- Cons: no automatic refactoring or version control GUI.

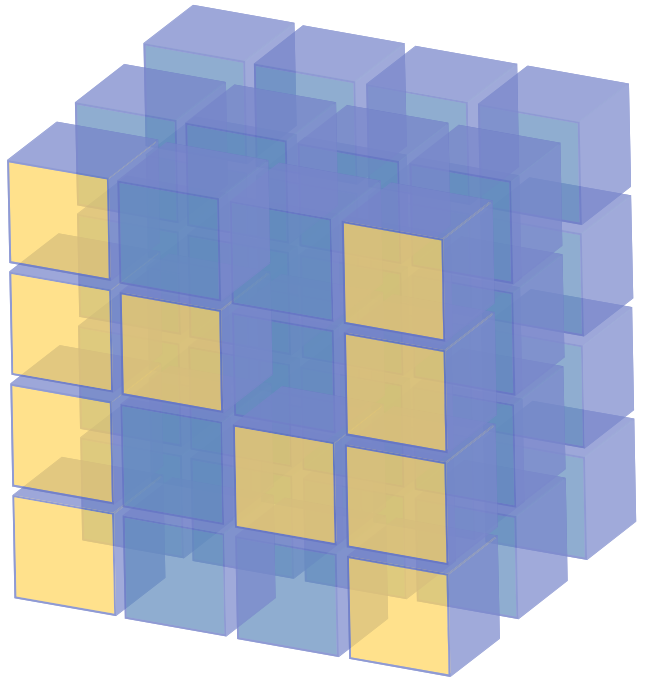
Libraries, accelerators

- Scientific Python consists of a number of separate libraries as usual with general-purpose programming languages
- **NumPy**, **SciPy**, **Matplotlib** the primary scientific libraries
- **SymPy** for symbolic computing
- **Numba** and **Cython** the most important *accelerators*
- Other, more specific libraries also available
  - See my Python course (lecture notes sec. 2; slide sets 5–8)

NumPy

http://www.numpy.org/

- *n*-dimensional arrays
- MATLAB style API, but instead of matrices, based on *cartesian tensors:*
  - A vector is a rank-1 tensor
  - A matrix is a rank-2 tensor



```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
A = np.array( [[1, 2],
               [4, 9]], dtype=np.float64 )
b = np.array( [3, 7], dtype=np.float64 )
x = np.linalg.solve(A, b)
```

SciPy

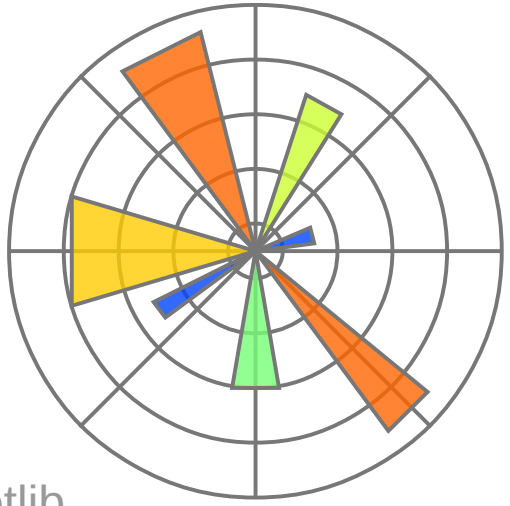
https://scipy.org/

- Advanced-user versions of linear algebra routines
- Sparse matrices
- I/O for MATLAB *.mat* files
- Numerical integration (quad), initial value problems (ODE), special functions
- Signal processing
- Some optimization solvers
- Cython interface to LAPACK, for advanced users

Matplotlib

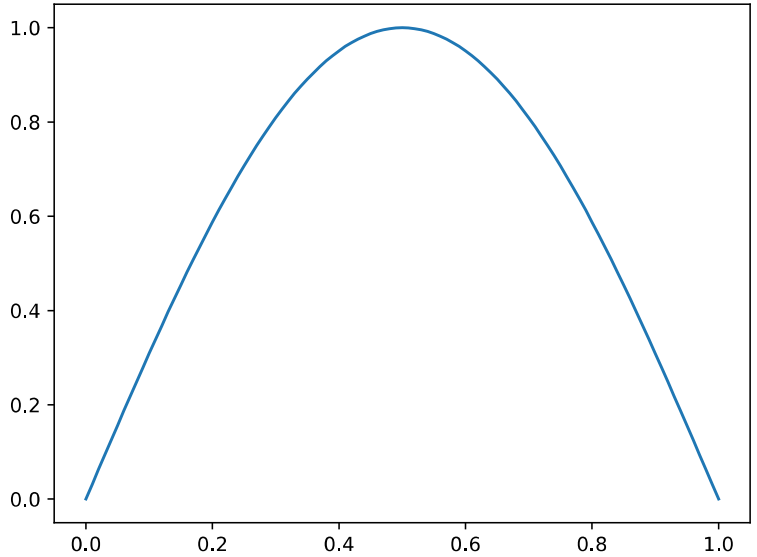
http://matplotlib.org/

- MATLAB-style plotting API
- Standard tool for publication-quality numerical graphics in Python



Also made with Matplotlib

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt
xx = np.linspace(0, 1, 101)
yy = np.sin(xx * np.pi)
plt.plot(xx, yy)
plt.savefig('sin_x.svg')
```

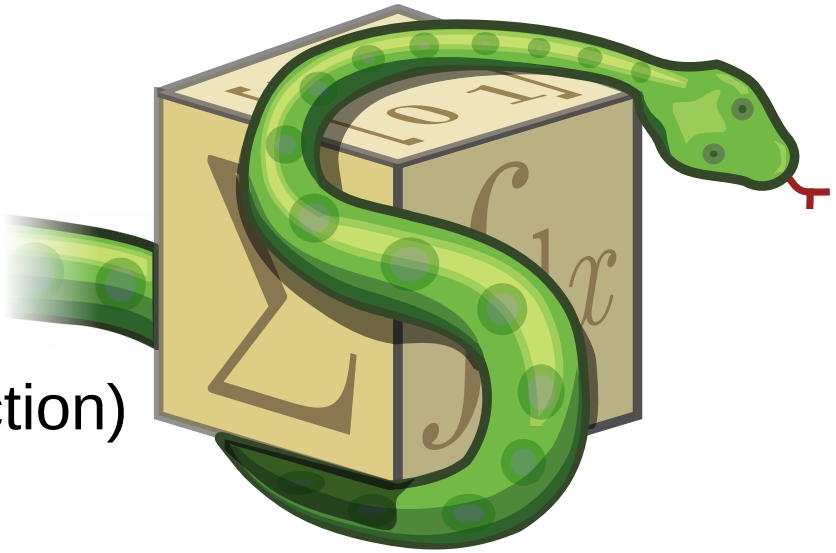


SymPy

http://www.sympy.org/

- Symbolic algebra, differentiation, integration

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sympy as sy
x = sy.symbols('x')
λf,λg = sy.symbols('f,g', cls=sy.Function)
g = λg(x)
f = λf(g)
D = sy.diff(f, x).doit()
sy.pprint(D)
```



- Python 3 allows Unicode variable names (with certain limitations).

```
  d                   d
──────(f(g(x)))⋅──(g(x))
dg(x)               dx
```

Input e.g. with a LaTeX input method

Numba

http://numba.pydata.org/

- Just-in-time (JIT) compiler for Python
- Accelerate functions by compiling them at runtime, when called for the first time.
- Supports a subset of Python; meant for accelerating data-crunching loops.


```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from numba import jit
from random import random

@jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random()
        y = random()
        if (x**2 + y**2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

Easy to use.

Cython

https://cython.org/

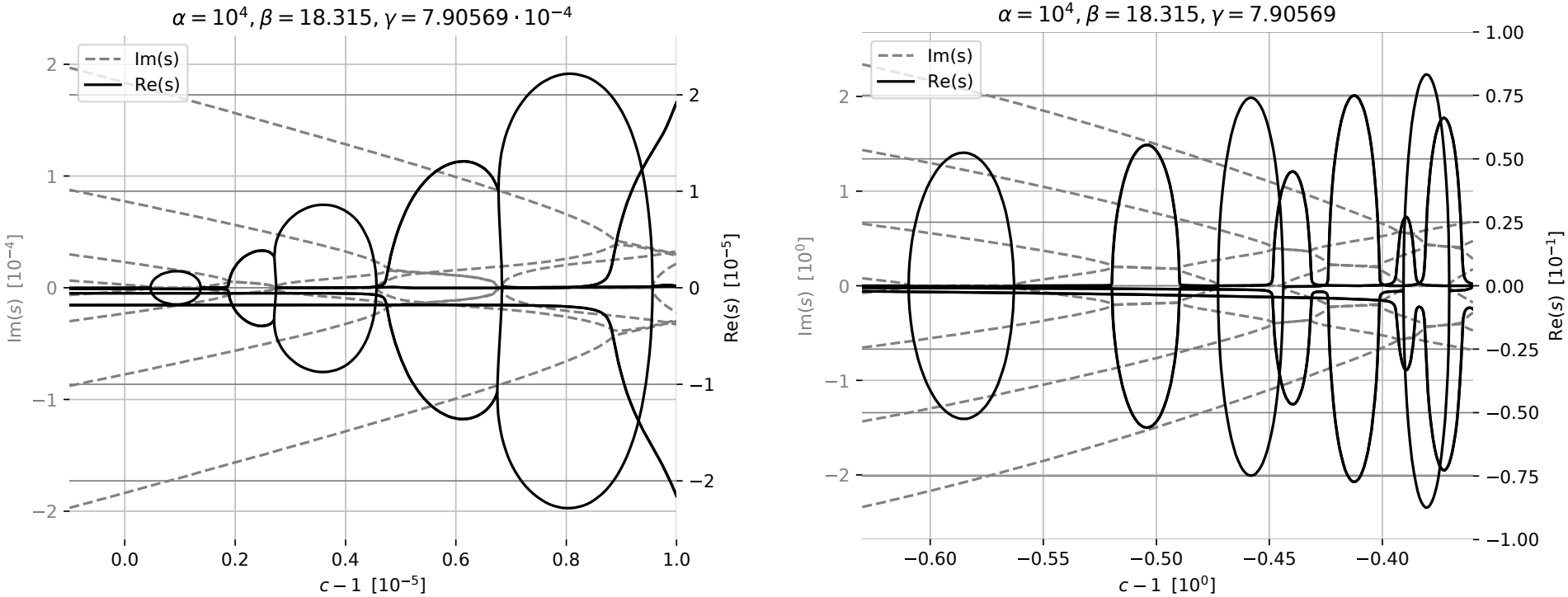
- Combine the power of Python and C
  - For implementing accelerated code, extends Python with a dialect of C that looks almost like Python
  - A superset of Python; allows mixing in Python wherever speed is not the primary concern
  - Easy to call to/from Python; compiles into Python extension modules that transparently interface with regular Python programs
- A language; detailed usage requires at least a full lecture.
- The other common use case of Cython is to create Python bindings for existing C libraries. For that use, see also CFFI and ctypes. For interfacing to Fortran instead of C, see F2PY.


```
def ddot(double [::1] a, double [::1] b):
    cdef unsigned int k
    cdef unsigned int n = a.shape[0]
    cdef double out = 0.0

    for k in range(n):
        out += a[k] * b[k]

    return out
```

What can we do with Python?

- **MATLAB-style numerics**:

- Stability exponents of an axially travelling viscoelastic Kelvin–Voigt panel subjected to a potential flow
- After discretization of the PDE, this is a quadratic eigenvalue problem
- After reduction to a generalized linear eigenvalue problem (via the companion form technique), solved using scipy.linalg.eig

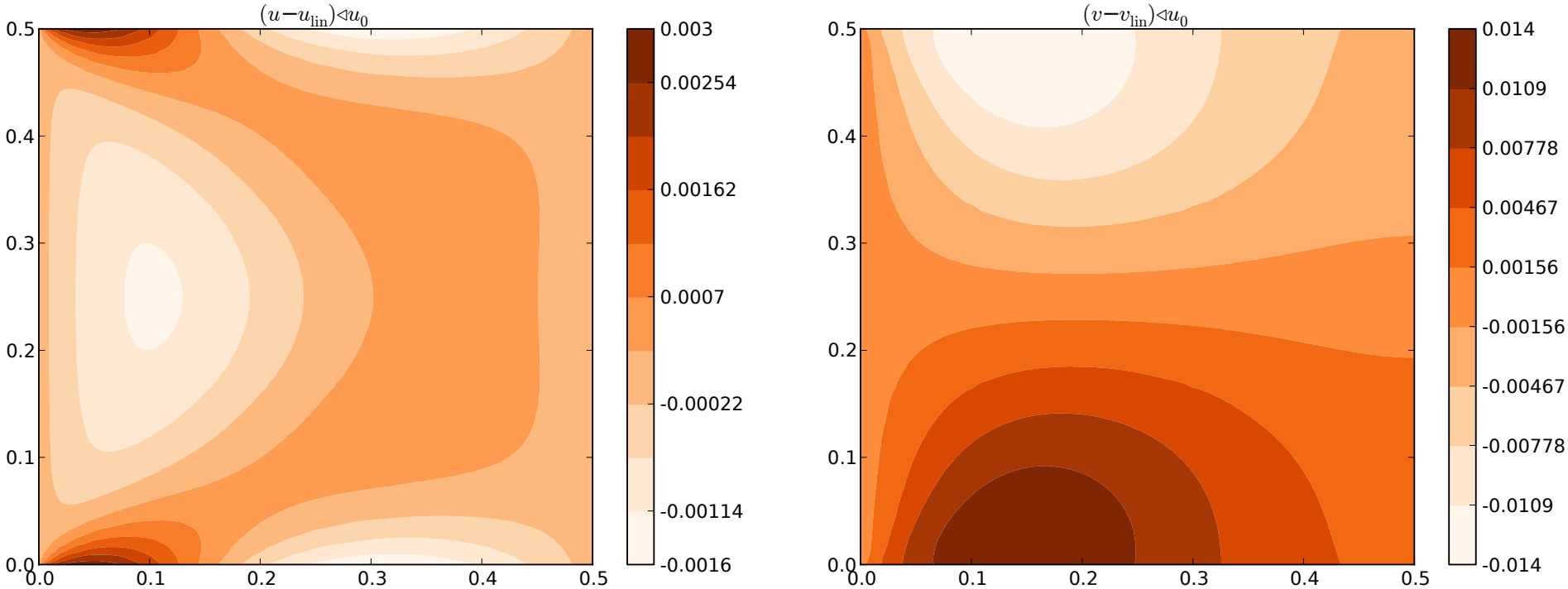


Stability of axially moving materials (to appear 2019, SMIA)

What can we do with Python?

- **MATLAB-style numerics**:

- In-plane deformation of an axially moving viscoelastic Kelvin–Voigt sheet
- Computed using a custom 2D C1 FEM code implemented in Python
  - **Actually recommended** FEM codes for Python: FEniCS Project, SfePy



10.1016/j.ijsolstr.2015.10.027

What can we do with Python?

- **Symbolic mathematics**:

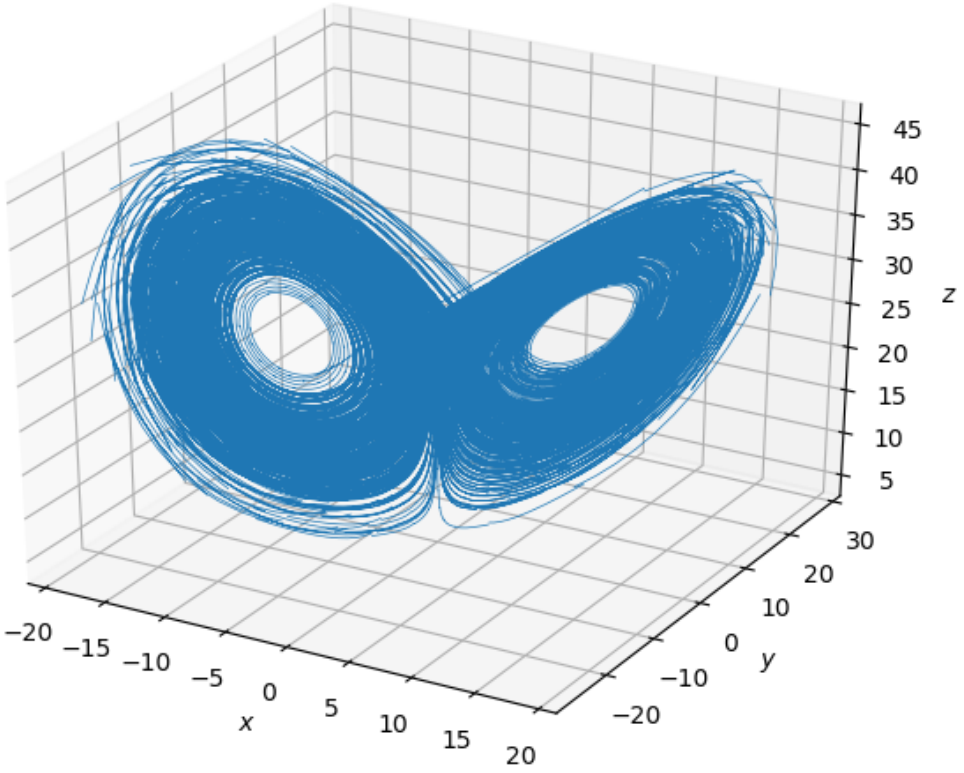
```
import sympy as sy

def hermite(k):
    """Derive C**k continuous Hermite interpolation polynomials for the interval [0, 1]."""
    order = 2*k + 1
    *A,x = sy.symbols('a0:{},x'.format(order + 1))

    w   = sum(a*x**i for i,a in enumerate(A))  # as a symbolic expression
    λw  = lambda x0: w.subs({x: x0})  # as a Python function; subs: symbolic substitution
    wp  = [sy.diff(w, x, i) for i in range(1, 1 + k)]  # diff: symbolic differentiation
    λwp = [(lambda expr: lambda x0: expr.subs({x: x0}))(expr) for expr in wp]  # why two lambdas: lecture notes sec. 5.8

    zero, one = sy.S.Zero, sy.S.One
    w0, w1 = sy.symbols('w0, w1')
    eqs  = [λw(zero) - w0, λw(one)  - w1]  # eqs. in form LHS = 0; see sympy.solve
    dofs = [w0, w1]

    for i, f in enumerate(λwp, start=1):
        d0_name = 'w{}0'.format(i * 'p')  # p = 'prime', to denote differentiation
        d1_name = 'w{}1'.format(i * 'p')
        d0, d1 = sy.symbols('{}, {}'.format(d0_name, d1_name))
        eqs.extend([f(zero) - d0, f(one)  - d1])
        dofs.extend([d0, d1])

    coeffs = sy.solve(eqs, A)
    solution = sy.collect(sy.expand(w.subs(coeffs)), dofs)

    N = [solution.coeff(dof) for dof in dofs]  # result: shape functions

    return tuple(zip(dofs, N))  # pairs (dof, shape function)
```

```
hermite(0)  # linear interpolation
((w0, -x + 1), (w1, x))

hermite(1)  # beam element
((w0, 2*x**3 - 3*x**2 + 1),
 (w1, -2*x**3 + 3*x**2),
 (wp0, x**3 - 2*x**2 + x),
 (wp1, x**3 - x**2))

hermite(2)  # 2nd derivative also continuous
((w0, -6*x**5 + 15*x**4 - 10*x**3 + 1),
 (w1, 6*x**5 - 15*x**4 + 10*x**3),
 (wp0, -3*x**5 + 8*x**4 - 6*x**3 + x),
 (wp1, -3*x**5 + 7*x**4 - 4*x**3),
 (wpp0, -x**5/2 + 3*x**4/2 - 3*x**3/2 + x**2/2),
 (wpp1, x**5/2 - x**4 + x**3/2))
```

What can we do with Python?

- **Implement and package algorithms**:
- **pydgq**: ODE system solver using dG(q), time-discontinuous Galerkin with a Lobatto (a.k.a. hierarchical) basis.

Lorenz system: $\rho = 28$, $\sigma = 10$, $\beta = 2.66667$, $x_0 = 0$, $y_0 = 2$, $z_0 = 20$



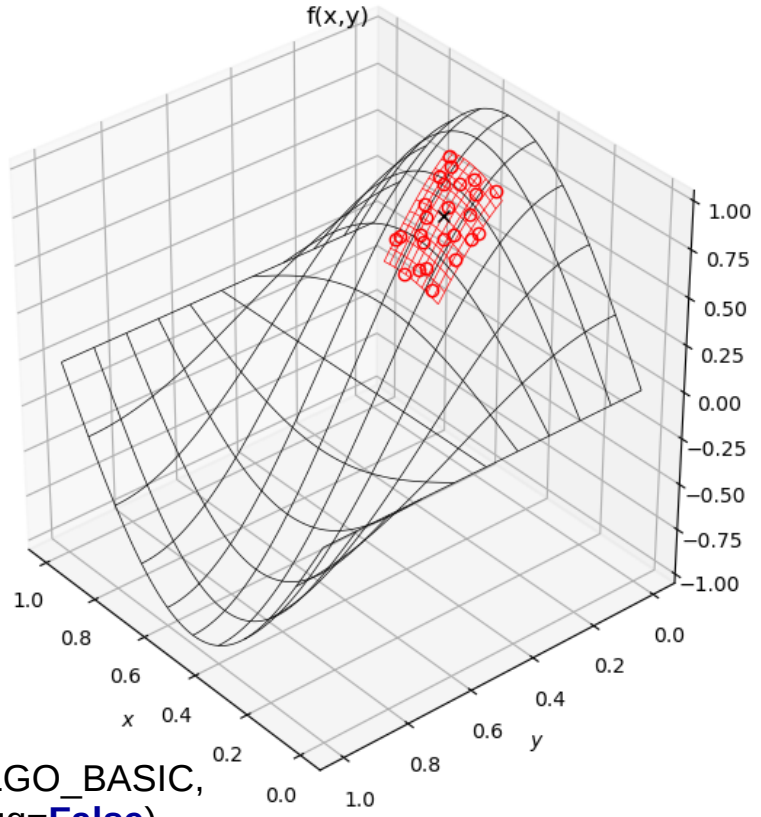

```
import numpy as np
from pydgq.solver.kernel_interface import PythonKernel
from pydgq.solver.galerkin import init
from pydgq.solver.types import DTYPE
from pydgq.solver.odesolve import ivp

class LorenzKernel(PythonKernel):
    def __init__(self, ρ, σ, β):
        super().__init__(n=3)
        self.p = [float(x) for x in ρ, σ, β]

    def callback(self, t):
        (ρ, σ, β), (x, y, z) = self.p, self.w
        self.out[:] = (σ*(y - x),  x*(ρ - z) - y,  x*y - β*z)

w0 = np.array([0, 2, 20], dtype=DTYPE)
rhs = LorenzKernel(ρ=28, σ=10, β=8/3)
init(q=2, method='dG', nt_vis=11, rule=None)
ww, tt = ivp(integrator='dG', interp=11, w0=w0, dt=0.1, nt=3500, rhs=rhs, maxit=10)
```

Full example on GitHub.

What can we do with Python?

- **Implement and package algorithms**:
- **wlsqm**: **w**eighted **l**east **sq**uares **m**eshless interpolator and differentiator

```
import numpy as np
from scipy.spatial import cKDTree as KDTree
import wlsqm

n, k, fit_order = 1000, 6, 2
f = lambda x: np.sin(np.pi*x[:, 0]) * np.cos(np.pi*x[:, 1])  # silly test data
x = np.random.random((n, 2))  # no mesh topology!
F = f(x)

tree = KDTree(data=x)  # Wikipedia: k-d tree
_, ii = tree.query(x, 1 + nk)
hoods = np.array(ii[:, 1:], dtype=np.int32)
kk = k * np.ones((npoints,), dtype=np.int32)
fit_orders = fit_order * np.ones((npoints,), dtype=np.int32)
knowns_bitmask = wlsqm.b2_F * np.ones((npoints,), dtype=np.int64)
wms = wlsqm.WEIGHT_UNIFORM * np.ones((npoints,), dtype=np.int32)
solver = wlsqm.ExpertSolver(dimension=2, nk=kk,
                            order=fit_orders, knowns=knowns_bitmask,
                            weighting_method=wms, algorithm=wlsqm.ALGO_BASIC,
                            do_sens=False, max_iter=10, ntasks=8, debug=False)
ndofs = wlsqm.number_of_dofs(dimension=2, order=fit_order)
fi = np.empty((npoints, ndofs), dtype=np.float64)
fi[:, 0] = F  # fi[i, 0] = function value at x[i, :]
solver.prepare(xi=x, xk=x[hoods])
solver.solve(fk=fi[hoods, 0], fi=fi, sens=None)
```



Full example on GitHub.

What can we do with Python?

- **Create a SymPy to Fortran compiler**:
- **mm-codegen**: Create material models for Elmer in SymPy
  - The compiler is < 2000 SLOC – **including** comments and docstrings!

m — s1 — s2 — p

Model

```
def let(k, v, s=True):
    defs[keyify(k)] = simplify(v) if s else v
Bs = sy.symbols("Bx, By, Bz")
εs = sy.symbols("εxx, εyy, εzz, εyz, εzx, εxy")
es = tuple(symutil.make_function(name, *εs)
        for name in ("exx", "eyy", "ezz", "eyz", "ezx", "exy"))
Is = tuple(symutil.make_function(name, *args)
                for name, args in (("I1", εs),
                                   ("I2", εs),
                                   ("I4", Bs),
                                   ("I5", Bs + es),
                                   ("I6", Bs + es)))
ψ = symutil.make_function("ψ", *Is)  # Helmholtz f.e.d.

B = sy.Matrix(Bs)                    # magnetic flux density
ε = symutil.voigt_to_mat(εs)         # Cauchy strain
e = symutil.voigt_to_mat(es)         # deviatoric strain
I1, I2, I4, I5, I6 = Is

ε_vol = sy.factor(sy.S("1/3") * ε.trace())
e_def = ε - ε_vol * sy.eye(3)
for _, (r, c) in symutil.voigt_mat_idx():
    let(e[r, c], e_def[r, c], s=False)
for key, val in ((I1, ε.trace()),
                 (I2, (ε.T * ε).trace())):
    let(key, val)
for key, val in ((I4, B.T * B),
                 (I5, B.T * e * B),
                 (I6, B.T * e * e * B)):
    let(key, val[0,0])  # unwrap the scalar
```

```
λ, μ = sy.symbols("λ, μ")
ψ_mech = sy.S("1/2") * λ * I1**2  +  μ * I2  # lin. elasticity

i0, nα, nβ, nγ = 1, 11, 1, 1
*αs, = sy.symbols("α{:d}:{:d}".format(i0, i0+nα))  # α1, .., α11
*βs, = sy.symbols("β{:d}:{:d}".format(i0, i0+nβ))
*γs, = sy.symbols("γ{:d}:{:d}".format(i0, i0+nγ))
I4_terms = sum(αi * I4**i for i, αi in enumerate(αs, start=i0))
I5_terms = sum(βi * I5**i for i, βi in enumerate(βs, start=i0))
I6_terms = sum(γi * I6**i for i, γi in enumerate(γs, start=i0))
ψ_magn = I4_terms + I5_terms + I6_terms

let(ψ, ψ_mech + ψ_magn)
```

- Separate *the functional dependency chains* from *the actual definitions* (important later, in **s1**).
- Can use SymPy for symbolic computation.
  - *Definitions, instructions; actual math automatic.*
- Still looks like a program, but *somewhat* close to the math (in the original Fortran spirit).
- Python 3 allows Unicode variable names.
  - For easily entering them, latex-input.
- *potentialmodelbase.py*, *polymodel.py* [minor edits]

m — s1 — s2 — p

The Elmer plugin

```
! s1 generated code (blank lines omitted for brevity):
REAL(KIND=dp) function I6(Bx, By, Bz, exx, exy, eyy, eyz, ezx, ezz)
use types
implicit none
I6 = Bx**2*(exx**2 + exy**2 + ezx**2) + 2*Bx*(By*(exx*exy + exy*eyy + &
    eyz*ezx) + Bz*(exx*ezx + exy*eyz + ezx*ezz)) + By**2*(exy**2 + &
    eyy**2 + eyz**2) + 2*By*Bz*(exy*ezx + eyy*eyz + eyz*ezz) + Bz**2* &
    (eyz**2 + ezx**2 + ezz**2)
end function


! corresponding s2 generated code (blank lines omitted for brevity):
REAL(KIND=dp) function I6_public(Bx, By, Bz, epsxx, epsxy, epsyy, epsyz, &
    epszx, epszz)
use types
implicit none
REAL(KIND=dp) exx_
! ...snip...
exx_ = exx(epsxx, epsyy, epszz)
exy_ = exy(epsxy)
eyy_ = eyy(epsxx, epsyy, epszz)
eyz_ = eyz(epsyz)
ezx_ = ezx(epszx)
ezz_ = ezz(epsxx, epsyy, epszz)
I6_public = I6(Bx, By, Bz, exx_, exy_, eyy_, eyz_, ezx_, ezz_)
end function
```

Generated from the model.

Generated from the **s1** code.

Jeronen, Rasilo, Kataja – A new material model for magnetostrictive materials in the open-source Elmer FEM software

See original presentation slides, Jeronen, XIII Finnish Mechanics Days, 2018.

Feature highlight: Lexical closures

- A feature of many high-level languages, with numerous applications. Has a long tradition in the Lisp family.
- Requires *first-class functions*. **Not supported** by Fortran or C. **Supported** by e.g. C++ 11, Java 8, Python, Racket and Clojure.
- Python example:

```
def make_adder(inc):
    def adder(x):
        return x + inc
    return adder
```

Calling **make_adder** causes the nested function **def**inition of **adder** to run, creating a new closure instance.

Here *inc* is a ***free variable*** (no local definition; not global).

We can then return the closure instance to the caller.

```
f = make_adder(inc=3)
g = make_adder(inc=17)
print(f(2))   # 5
print(g(25))   # 42
```

The ***closure property*** is that an instance permanently retains access to the *inc* that was *passed in by the caller of* **make_adder** when that closure instance was created.

(The name *closure* means that a surrounding, non-global scope ***closes over*** the free variables of the inner scope.)

- Eliminate boilerplate. Define local helper functions locally, in a nested **def**. In the inner definition, list as parameters only those that actually vary.
- Separate public interface from implementation compactly, without exposing internal details (example later).
- Change how existing functions behave; see Python's **decorators** [1] [2]
- Create an object system (in Python, don't do that – it already has one!).
- Create a continuation system. (See a simple explanation of continuations.)

Advanced: Syntactic macros

Scheme code is not meant to be written by humans, [but] … automatically by macros.
–Michele Simionato

- Syntactic macros are an advanced feature *almost* unique to the Lisp family. Exceptions: Julia, R.

abstract syntax tree

- **What**: **Syntactic macros** transform the AST, by running arbitrary code on it, at compile time.
  - Contrast C preprocessor macros, which perform only text substitution.
  - Contrast C++ generics: *Lisp itself as metalanguage* (no separate templating mini-language).
  - StackOverflow. Understanding macros. Perl perspective. Macros and washing machines.
  - Paul Graham (1993): Programming bottom-up; Metaprogramming; Extensible programming.
  - *Lisp isn't a language, it's a building material. –Alan Kay* (of Smalltalk fame; on Lisp, [1] [2])

- **Why**: *Design patterns: a symptom of being unable to extract an abstraction* (Paul Graham).
  - E.g. **with** [1] or **assert** [2] in Python, which encode particular design patterns.
  - Syntactic macros allow *the programmer* to create such constructs:
    - **with** in Clojure.
    - Delayed evaluation in Racket.
  - Just like in mathematics [1] [2]: code is for humans, so ***notation matters***.
  - Macros are an important feature that make **Lisp feel more like a fluid than a solid**.
  - Democratization of language design? On the other hand, herd of cats (according to some, with machine guns), no BDFL. No process to pick, polish and promote the best abstractions; thus, “lowest common denominator” often used. *The Lisp Curse* [1] [2].

Advanced: Syntactic macros

- **The nuclear option**: only create a macro if the job is not suitable for a run-of-the-mill function!
  - *Extract design patterns that **cannot** be extracted as functions*.
    - Although Racket is eager, macro arguments avoid immediate evaluation; highly useful [1][2].
    - ***Macros replacing design patterns*** is what "programs writing programs" means in Lisp; it's not about "source code generation" à la Cython (which takes Cython and writes C).
      - It's also robust, unlike source filters in many languages.
  - Add syntactic forms the original designer of the language might not approve. [1]
  - Create a ***DSL*** (*domain-specific language*) to fit the language to your domain; shorter code.
  - DRY out repetition in a set of similar macros, by *macro-writing macros*. (Example.)
  - Programmatically create lookup tables at compile time. [1]

- **Limitations**:
  - **Second-class**; cannot pass a macro as an argument: expanded away at compile time!
  - **Local**: a macro call cannot rewrite any forms *surrounding* it (due to Lisp's prefix notation)
  - Macros cannot change the lexical conventions (use of parentheses, prefix notation, …)
    - If you want to do that, you could modify or extend the ***reader*** [1] [2].
  - **Macros don't compose**. In a multi-layered macro library, each layer needs to know about all of the previous layers. This build-up of complexity limits what can be achieved in practice.
  - Still, Racket itself is built mostly from macros; very few primitives in a fully expanded program!

- Examples in Racket: Non-deterministic evaluation, automatic currying, Python-inspired syntactic forms, simple infix math, User-programmable infix operators, Algebraic Data Types (ADTs) in Typed Racket.

- MacroPy: Syntactic macros for Python.

What can we build with MacroPy?

Let constructs like those in the Lisp family and Haskell.

Bind names for the duration of one expression. Use cases:

- Break an expression into easily readable chunks, without polluting the surrounding scope with temporaries.
- Explicitly indicate which definitions are needed only locally.

```
from unpythonic.syntax import macros, let, letseq, letrec

let[((x, 17),  # parallel binding, i.e. bindings don't see each other
     (y, 23)) in
       print(x, y)]

letseq[((x, 1),  # sequential binding, i.e. Scheme/Racket let*
        (y, x+1)) in
         print(x, y)]

# mutually recursive binding, sequentially evaluated
t = letrec[((is_even, lambda x: (x == 0) or is_odd(x - 1)),
            (is_odd, lambda x: (x != 0) and is_even(x - 1))) in
           is_even(42)]
```

Automatic tail call optimization (TCO):

```
from unpythonic.syntax import macros, tco

with tco:
    is_even = lambda x: (x == 0) or is_odd(x - 1)
    is_odd  = lambda x: (x != 0) and is_even(x - 1)
    assert is_even(10000) is True

with tco:
    def is_even(x):
        if x == 0:
            return True
        return is_odd(x - 1)
    def is_odd(x):
        if x != 0:
            return is_even(x - 1)
        return False
    assert is_even(10000) is True
```

Macros on this and the next slide are available in **unpythonic**, and they mostly work together. See documentation.

What can we build with MacroPy?

Continuations (call-with-current-continuation a.k.a. call/cc):

```
from unpythonic.syntax import macros, continuations, call_cc
with continuations:
    stack = []
    def amb(lst, cc):  # McCarthy's amb operator
        if not lst:
            return fail()
        first, *rest = tuple(lst)
        if rest:
            ourcc = cc
            stack.append(lambda: amb(rest, cc=ourcc))
        return first
    def fail():
        if stack:
            f = stack.pop()
            return f()

    def pt():  # Pythagorean triples
        z = call_cc[amb(range(1, 21))]
        y = call_cc[amb(range(1, z+1))]
        x = call_cc[amb(range(1, y+1))]
        if x*x + y*y != z*z:
            return fail()
        return x, y, z
    t = pt()
    while t:
        print(t)
        t = fail()  # ...outside the dynamic extent of pt()!
```

Automatic currying:

```
from unpythonic.syntax import macros, curry
from unpythonic import foldr, composerc, cons, nil

with curry:
    def add3(a, b, c):
        return a + b + c
    assert add3(1)(2)(3) == 6

    # see John Hughes: Why FP Matters
    my_map = lambda f: foldr(composerc(cons, f), nil)
    double = lambda x: 2 * x
    print(my_map(double, (1, 2, 3)))
```

Call-by-need functions:

```
from unpythonic.syntax import macros, lazify

with lazify:
    def g(a, b):
        return a  # b is never used
    def f(a, b):
        return g(2*a, 3*b)
    assert f(21, 1/0) == 42
```

…this is starting to look like a custom language?

Packaging a language: *Dialects*

- **Pydialect** implements a dialect system, in pure Python, based on *import hooks* [1] [2].
  - Motivation: Language semantics and surface syntax encode patterns at a very high level.

- Disclaimer: **Very much** outside the vision for the official Python language.
  - PEP 511 was rejected for the specific reason it could be seen as officially blessing the creation of dialects.
  - The native habitat of this idea is Racket (*Solve problems. Make languages.*).

```
from __lang__ import lispython

def fact(n):
    def f(k, acc):
        if k == 1:
            return acc
        f(k - 1, k*acc)
    f(n, acc=1)
assert fact(4) == 24
fact(5000)
```

__lang__ is a magic module that doesn't actually exist. Importing a dialect from it triggers the dialect processor **when the program is run with `pydialect`** instead of bare Python. Dialects may define new surface syntax and/or change semantics.

Public interface

Implementation

Implicit **return** in tail position

Tail call optimization; O(1) call stack depth with tail calls, so tail calls can be used for recursion-based looping, like in Scheme and in Racket.

```
t = letrec[((is_even, lambda x: (x == 0) or is_odd(x - 1)),
            (is_odd, lambda x: (x != 0) and is_even(x - 1))) in
           is_even(10000)]
assert t is True

g = lambda x: [local[y << 2*x],
               y + 1]
assert g(10) == 21
```

⚠ This code is not Python; it's Lispython.

Literature

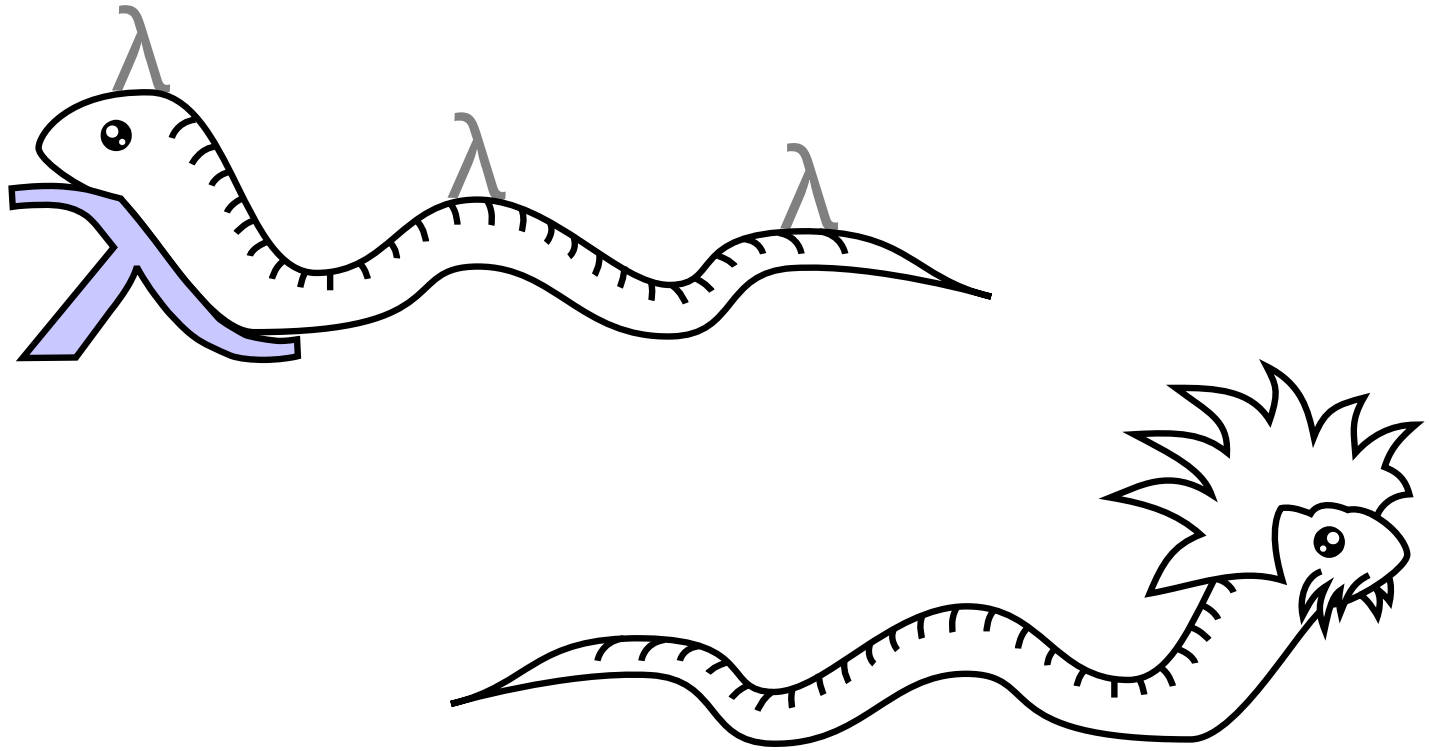
- Mark Lutz: *Learning Python*, 5th ed., O'Reilly, 2013.
  - Standard “bible” of the trade, very comprehensive.
  - A couple of minor versions behind the latest Python.

- Luciano Ramalho: *Fluent Python: Clear, Concise and Effective Programming*, O'Reilly, 2015.
  - Python 3 for programmers coming from other languages. Focuses on features that are easily missed, if the reader is used to thinking in another programming language.

- Zed A. Shaw: *Learn Python 3 the Hard Way*, Addison–Wesley, 2017.
  - For newcomers to programming.
  - *Hard way* because *There is no royal road to geometry. –Euclid*

- Internet!
  - Especially Stack Overflow.
  - For a self-contained introduction to Python in scientific computing:
    - Python 3 for scientific computing, my course held at TUT, 2018; covers also background in CS/IT; see esp. slides and exercises.
    - Scipy Lecture Notes, a community-maintained course with a tight focus on the scientific computing parts only.

The finish line



Thank you for your attention!