TAMkin from scratch

T. Verstraelen

Center for Molecular Modeling, Ghent University, Belgium

Toon.Verstraelen@UGent.be

Center for Molecular Modeling

Belgian Science Policy Office

belspo

fwo

UNIVERSITEIT GENT

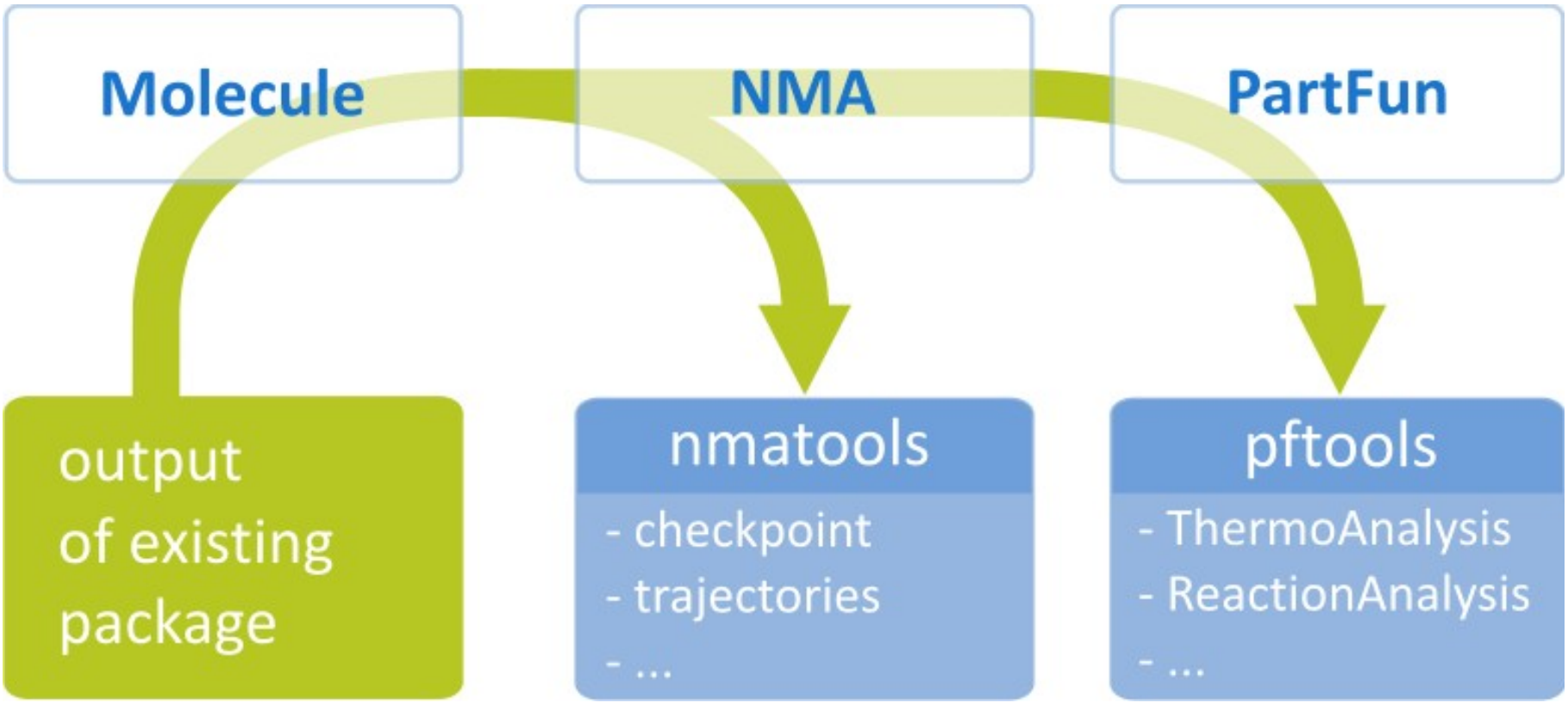
Outline

TAMkin Overview

Python (2.7) Primer

TAMkin (1.0) examples

Flowchart



Output (Hessian, geometry, ...) from different simulation codes can be read:
- Gaussian
- CP2K
- ORCA
- ...

Customization of normal mode analysis:
- frozen atoms
- rigid blocks
- ...

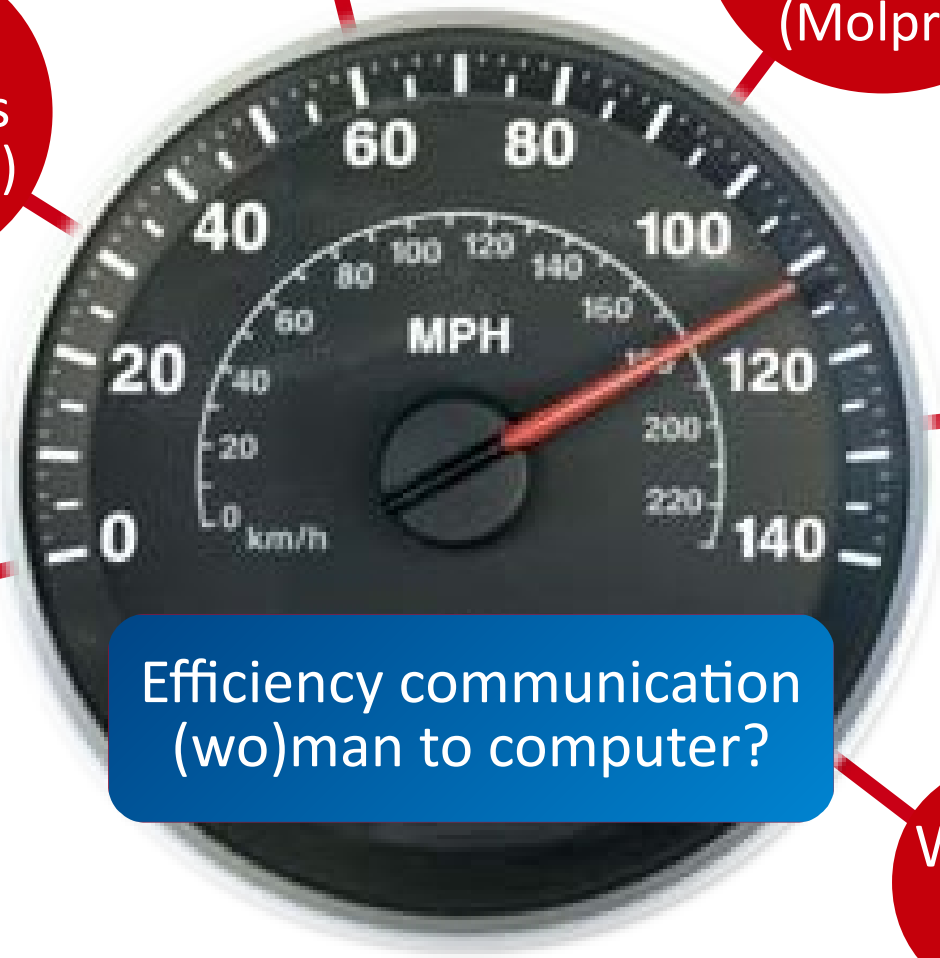
Customization of partition functions:
- temperature, pressure
- internal rotors
- ...

Chemical equilibrium
Reaction kinetics (tunneling)

Important URLs

- TAMkin homepage
  http://molmod.github.io/tamkin
- MolMod homepage (general molecular modeling tools, used by TAMkin)
  http://molmod.github.io/molmod
- CMM software page
  http://molmod.ugent.be/software
- Browse source code (history) of TAMkin
  https://github.com/molmod/tamkin

TAMkin input (1)



Flexible input files (CP2K)

Command file with language constructs (Molpro)

a input files (Gaussian)

Full-featured scripting language + a modeling library

Punch cards

Efficiency communication (wo)man to computer?

What is next?

TAMkin input (2)

TAMkin = Python package (software library with well-designed API)

TAMkin input = Python script that uses the TAMkin package

→ Basic knowledge of Python required.

Outline

TAMkin Overview

Python (2.7) Primer

TAMkin (1.0) HowTo

Scripting versus compiled

Scripting languages (interpreted)

- Bash
- Python
- Perl
- Matlab / Octave
- Basic
- TCL

...

Faster development
Slower execution
More features
No compilation

(Byte-)Compiled languages

- C
- C++
- Fortran
- Cython
- Basic
- Pascal
- Java(script)

...

Slower development
Faster execution
Less features
Compilation

General background

Brief History

1994

- First release (1.0)
- Main developer: Guido Van Rossum (BDFL)
- Influences from lisp, ABC, C++, ...
- Small community (not popular)

2000

- Influential release (2.0)
- Mature language, easier to use
- Open community, widespread adoption
- Latest in this series 2.7.x (most popular version)

2008

- Ideological cleanup release (3.0)
- Remove redundancies: less features
- Fix design mistakes
- Not backward compatible
- Slow adoption

General background

Multiparadigm

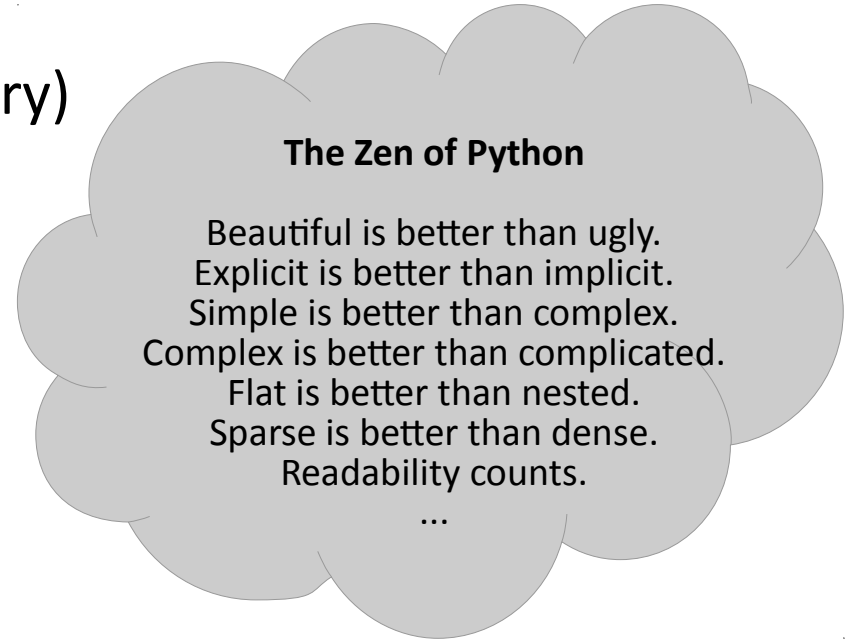
- Object-oriented programming
- Functional programming (lambda calculus)
- Procedural programming
- Structured programming
- ...

Distinctive features

- Light syntax (unlike perl, bash, ...)
- High-level built-in datatypes (sets, dictionaries, ...)
- Dynamic typing
- Garbage collection
- Batteries included (large “standard” library)
- Many third-party libraries
- Late binding

Ideology: when something is *pythonic*

- Readability (indentation, doc strings)
- There is only one way to ...
- The Zen of Python



Useful Python links

The website
http://www.python.org/

Python Tutorial from zero to hero
http://docs.python.org/tutorial/index.html

Reference of standard libraries
http://docs.python.org/library/index.html

Extensive Q&A site
http://stackoverflow.com/

Numpy/Scipy/Matplotlib
http://www.scipy.org/

Python package database
http://pypi.python.org/pypi

Python 2.X is installed by default on any (recent) Linux distro

Interactive usage

- Start the `python` program.
- You'll get the Python command line:

```
Python 2.7.5 (default, Feb 19 2014, 13:47:28)
[GCC 4.8.2 20131212 (Red Hat 4.8.2-7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Enter code, line by line, e.g. `1 + 1`

Python scripts

- Create a file, e.g. script.py
- First line: `#!/usr/bin/env python`
- Comments start with hash: #
- Add some lines of Python code
- Make executable (once) and run:

```
chmod +x script.py
./script.py
```

- Or execute directly:

```
python ./script.py
```

Built-in datatypes include:

- bool: True or False

```
print 2 == 4
```

- int, long: integer

```
print 2 + 2
print 3/2           # integer division!
```

- float, complex: floating point number (double precision)

```
print 2.0*2.0
print 2.0**3
```

- str: character string

```
'abc', "abc"        # single and double quotes are allowed
'''This is
a multiline
string'''
```

- list & tuple: a list of objects (mixed datatypes allowed)

```
l = [1, 2, 3]
t = (1, 2, 3)
l.append(4)         # append is a “method” of the list type
print l
print l*3
print l[0]          # print only the first element, which has index 0
```

- dict: a dictionary or associative array (mixed datatypes allowed)

```
d = {1: 5, 'a': 7, 'b': 'c', 'c': 3} # {key1: value1, key2: value2, ...}
print d[1]          # square brackets are also used to select element/slice
del d['b']          # deletes key-value pair 'b': 'c'
d['foo'] = 'bar'    # add new association
print d
```

- set: an unordered list with unique elements (mixed datatypes allowed)

```
s1 = set([1, 3, 'a'])
s2 = set([5, 1, 'b'])
print s1 & s2, s1 | s2
```

Mutable versus Immutable datatypes

Only a handful of datatypes are immutable.

- Built-in: bool, int, long, float, complex, str, tuple, bytes, frozenset
- Contents of the object can not be changed after creation.
- Object can only be replaced.
- Keys in dictionaries and elements in sets must be immutable.

Nearly all other types are mutable.

- Contents can be changed after creation.
- Examples: list, dict, set, ...

Demo

```
# lists are mutable, one can add/delete/change elements after creation
l = [1, 2, 3]
l.append(4)
l[0] = 0              # replaces first element by 0
 del l[1]

# tuples are immutable
t = (1, 2, 3)
t.append(4)           # does not work
t[0] = 0              # does not work
print 2*t             # works! 2*t creates a new tuple
print t + (4, 5, 6)   # works! t + (4, 5, 6) creates a new tuple
```

All variables are references to objects.

- Everything is allocated dynamically
- No explicit deallocation needed (garbage collection)

Demo

```
l1 = [1, 2, 3]
l2 = l1
l2.append(4)
print l1        # l1 also contains 4, l1 and l2 refer to same object
print l1 == l2  # True
print l1 is l2  # True

l3 = [1, 2, 3, 4]
print l1 == l3  # True
print l1 is l3  # False
```

String formatting (Python 2.X style)

- General structure `'... %X ... %X' % (var1, var2)`
- The *X* in %*X* determines the formatting of the variable:
  - %i: integer.
  - %s: string (or anything that can be converted to a string)
  - %*x*.*y*f: where x is the width of the number and y the number of decimals after the dot.
  - %*x*.*y*e: same but with mantissa (scientific notation)
  - ...
  - For a complete overview: https://docs.python.org/2.7/library/stdtypes.html#string-formatting-operations

Demo

```
x = 1.25
i = 2
print '%5.3f times %i equals %5.3f' % (x, i, x*i)
```

Conditionals: “if”, “elif”, “else”

- General structure:

```
if condition1:
    line 1 in if block
    line 2 in if block
    ...
elif condition2:          # optional
    line 1 in elif block
    line 2 in elif block
    ...
elif condition3:          # optional, multiple elif blocks allowed
    ...
else:                     # optional
    line1 in else block
    line2 in else block
    ...
first line after if-elif-else blocks
```

- Indentation matters!
- Indentation = 4 spaces by convention (not mandatory)
- Indentation expected after line that ends at colon.

Loops: "while"

- General structure:

```
while condition:
    line 1 in while block
    line 2 in while block
    ...
first line after loop
```

- The "condition" is an expression that is True or False.

Demo

```
# Fibonacci numbers below 20
i1 = 1
i2 = 1
while i2 < 20:
    print i2
    tmp = i1
    i1 = i2
    i2 = tmp + i1
print 'done'
```

```
# Alternative implementation
i1 = 1
i2 = 1
while i2 < 20:
    print i2
    i1, i2 = i2, i1+i2

print 'done'
```

Loops: “for”

- General structure:

```
for var in iterator:
    line 1 in for block
    line 2 in for block
    ...
first line after for block
```

- An iterator is needed: list, tuple, dict, set, ...
- xrange(begin, end, step) = iterator over integers: begin, begin+step, begin+2*step, ... < end. Default arguments:
  - begin = 0
  - step = 1

Demo

```
# Truncated taylor series of exponential function of x=0.357
x = 0.357
exp = 0.0
factorial = 1
for i in xrange(5):
    if i > 0:
        factorial *= i
    exp += x**i/factorial
```

$$\exp(x) \approx \sum_{n=0}^{N} \frac{x^n}{n!} + O(x^{n+1})$$

functions: “def”

- General structure:

```
def function_name(arg1, arg2, ... optarg1=x, optarg2=y, ...):
    line 1 in function block
    line 2 in function block
    ...
    return returnvariable # optional, allowed anywhere in function
first line after function block
```

- When return is not used (reached), the `None` constant is returned.
- Multiple return values are possible: `return a, b, c` -> returns a tuple.
- Functions are called as follows: `function(arg1, arg2, ...)`

Demo

```
# Recursive factorial function
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n*factorial(n-1)
print factorial(7)
```

Moving on ...

Python modules

- A file with extension .py
- Does not (have to) start with `#!/usr/bin/env python`
- Contains Python code that can be *imported*.

common.py

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n*factorial(n-1)
```

program1.py

```
#!/usr/bin/env python
from common import *
print factorial(7)
```

program2.py

```
#!/usr/bin/env python
from common import factorial
print factorial(7)
```

program3.py

```
#!/usr/bin/env python
import common
print common.factorial(7)
```

Moving on ...

Python packages

- A directory with .py-files and a file __init__.py
- Represents a collection of modules
- Can be hierarchical (subdirectories -> subpackages)
- Contains Python code that can be *imported*.

common/__init__.py (empty)

common/factorial.py

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n*factorial(n-1)
```

program.py

```
#!/usr/bin/env python
from common.factorial import *
print factorial(7)
```

Moving on ...

Object-oriented programming

- This is a big topic, impossible to cover completely in one day.
- Very powerful!
- Extensively used in TAMkin
- Purpose
  - Formally: define new datatypes
  - Conceptually: hide complexity

Object-oriented programming

```
#!/usr/bin/env python

class Car(object):
    def __init__(self):
        self.speed = 0.0

    def accelerate(self):
        self.speed = self.speed + 5.0


corvette = Car()
lada = Car()

print corvette.speed   # 0.0
corvette.accelerate()
print corvette.speed   # 5.0

lada.speed = 3.0
print lada.speed       # 3.0
```

A class definition with two methods

__init__: special case, also called constructor

self: special variable, reference to instance

create instances of Car class, call the constructor method and assign to variable result

```
corvette = Car.__new__(Car)
Car.__init__(corvette)
```

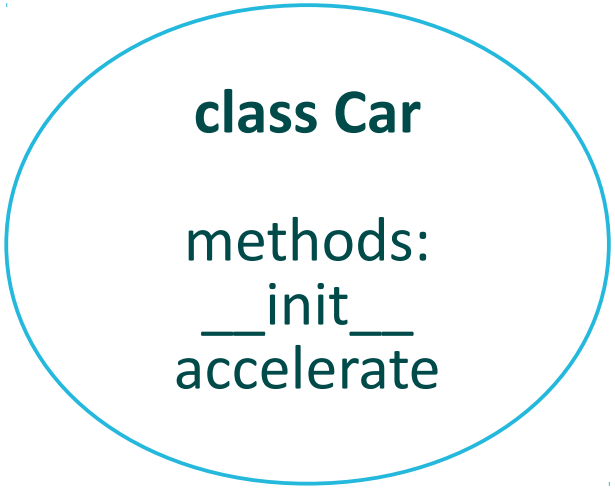
speed is an attribute of every Car instance.

accelerate is a method of every Car instance.

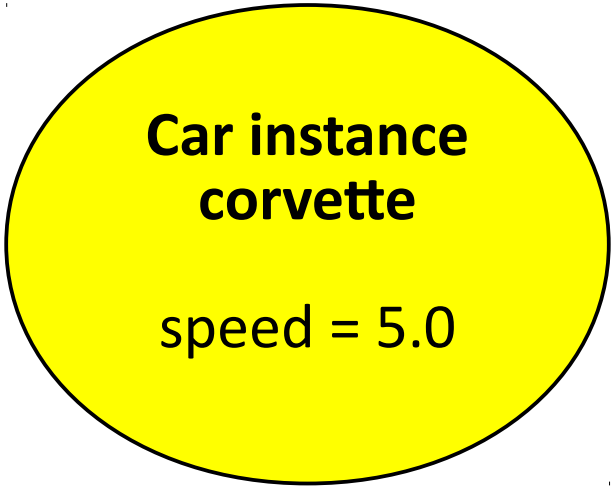
```
Car.accelerate(corvette)
```

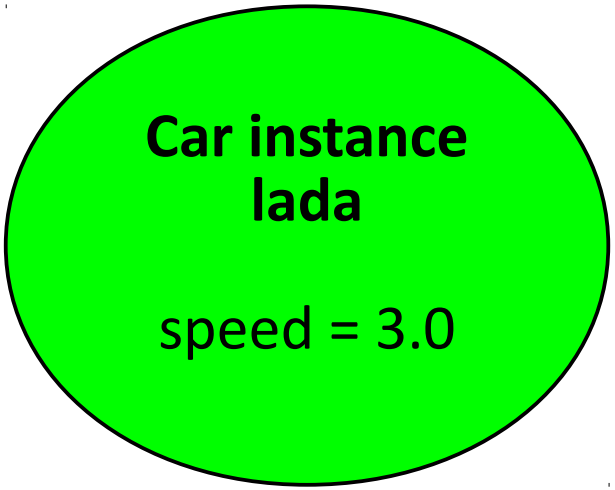
Moving on ...

Class =
blue print of the objects



Instances/Objects =
actual things in computer memory





...

Moving on ...

Exceptions

- Mechanism to report/handle errors
- Exceptions can be “caught” to handle the error
- When an exception is not “caught”:
  - In a script: error message + the program stops
  - In interactive mode: error message

Demo (interactive session)

```
Python 2.7.5 (default, Feb 19 2014, 13:47:28)
[GCC 4.8.2 20131212 (Red Hat 4.8.2-7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 2.0/0.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
>>> l = [1, 2]
>>> l[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

Moving on ...

Exceptions

- Mechanism to report/handle errors
- Exceptions can be “caught” to handle the error
- When an exception is not “caught”:
  - In a script: error message + the program stops
  - In interactive mode: error message

exception.py

```
#!/usr/bin/env python
def divide(x, y):
    return x/y
print divide(2.0, 0)
```

When executing this script on the command line:

```
chmod +x exception.py
./exception.py
Traceback (most recent call last):
  File "./exception.py", line 4, in <module>
    print divide(2.0, 0)
  File "./exception.py", line 3, in divide
    return x/y
ZeroDivisionError: float division by zero
```

Moving on ...

“Doc strings” -> inline documentation

- String as first *line* of a function, class or module, e.g.:

```
def function(a, b, c):
    '''Solves the quadratic equation if the solutions are real'''
    d = b**2 - 4*a*c
    if d > 0:
        return (-b+d**0.5)/(2*a), (b--d**0.5)/(2*a)
    elif d == 0:
        return -b/(2*a), -b/(2*a)
```

- This is used extensively in many Python packages (including TAMkin).
- Documentation is often generated from the doc strings, e.g.: http://molmod.github.io/tamkin/reference/io.html

Coding style

- Conventions to improve readability.
- Full guideline (PEP8): http://legacy.python.org/dev/peps/pep-0008/
- The basics (from the official Python tutorial):
  - Use 4-space indentation, and no tabs.
  - Wrap lines so that they don't exceed 79 characters.
  - Use blank lines to separate functions and classes, and larger blocks of code inside functions.
  - When possible, put comments on a line of their own.
  - Use docstrings.
  - Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
  - Name your classes and functions consistently; the convention is to use CamelCase for classes and lower_case_with_underscores for functions and methods. Always use self as the name for the first method argument.

Important & Popular Scientific Python Packages

NumPy

- Dense arrays of numerical data
- Efficient array operations (linear algebra, vectors and matrices)
- Packages available for all major Linux distro's (not installed by default)
- Uses FORTRAN BLAS and LAPACK internally.

Demo (interactive mode)

```
>>> import numpy as np
>>> a = np.array([[1.0, 1.0], [0.0, -1.0]])
>>> print a
[[ 1.  1.]
 [ 0. -1.]]
>>> b = np.array([2.0, 3.0])
>>> print b
[ 2.  3.]
>>> print np.dot(a, b)     # matrix-vector product
[ 5. -3.]
>>> print a[0,1]           # element at first row and second column
1.0
>>> print a[0]             # first row
[ 1.  1.]
>>> print a[:,1]           # second column. slice format [begin]:[end][:step]
[ 1. -1.]
```

Matplotlib

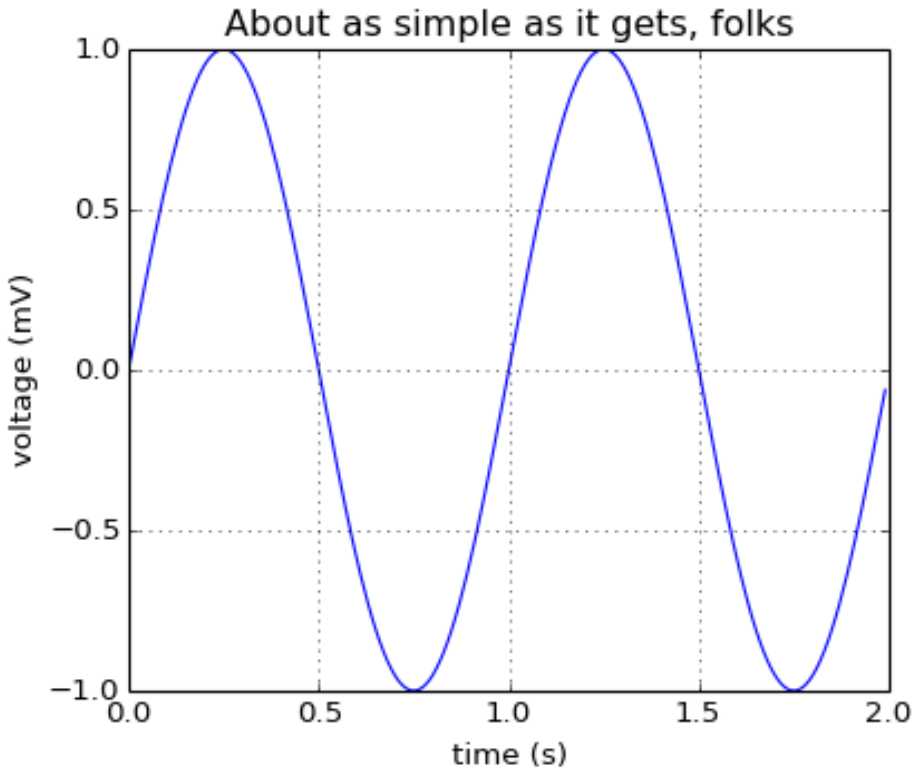
- Scientific plotting library for Python
- Usage is very similar to Matlab plotting functions
- Packages available for all major Linux distro's (not installed by default)

Demo (adapted from matplotlib website)

```
import matplotlib.pyplot as pt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*pi*t)
pt.plot(t, s)

pt.xlabel('time (s)')
pt.ylabel('voltage (mV)')
pt.title('About as simple as '
         'it gets, folks')
pt.grid(True)
pt.savefig("test.png")
```

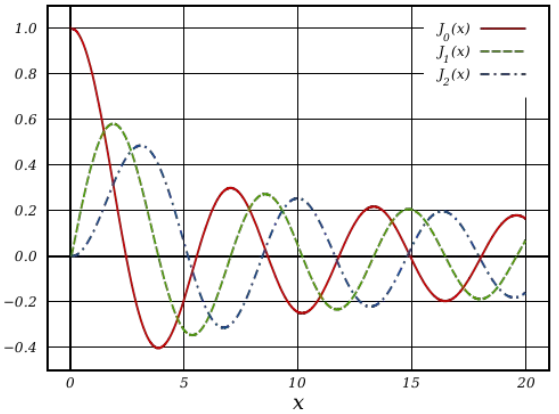


SciPy

- Extends Numpy with more diverse and specialized tools:
  - function minimzation
  - sparse matrices
  - special functions
  - ...
- Packages available for all major Linux distro's (not installed by default)

Demo (interactive mode)

```
>>> from scipy.special import j0        # Bessel function, 1st kind, order0
>>> from scipy.optimize import minimize # General purpose numerical minimizer
>>> result = minimize(j0, 4)            # Initial guess is 4
>>> result.x
array([ 3.83170597])
>>>
```



Outline

TAMkin Overview

Python (2.7) Primer

TAMkin (1.0) HowTo

Units

Internally: atomic units (and Kelvin for temperature)

- Consistent units (like SI)
- Reasonable numbers in atomic simulations

Converting to internal units

```
>>> from molmod import *
>>> l = 5*angstrom          # convert 5 A to atomic units
>>> energy = 10*kjmol       # convert 10 k J mol^-1 to atomic units
```

Convertion from internal units

```
>>> print energy            # energy in hartree
0.00380879917604
>>> print energy/kcalmol    # energy in k cal mol^-1
2.39005736138
```

How to run Gaussian *frequency* jobs for TAMkin

Mandatory: compute the Hessian (properly)

- Add `opt(tight)` to input (well-optimized geometries improve reproducibility)
- In case of DFT, add `int(grid=ultrafine)` to improve accuracy.
- In case of G03, add `scf(tight)`, to improve accuracy. This is the default in G09.
- Add `freq(noraman)` to input (to compute the Hessian, skipping Raman intensities to save 10% of CPU time)

Mandatory: keep the *formatted* checkpoint file (contains Hessian)

- Add line to input: `%chk=somefile.chk`
- After job completion: `formchk somefile.chk && rm somefile.chk`
- Note that frequencies from log file are not used.

Recommended: organize your files

- A separate directory for every Gaussian job
- Use the same name for each type of file: `gaussian.com gaussian.log gaussian.chk guassian.fchk`. This facilitates automation.

How to run Gaussian *relaxed scan* jobs for TAMkin

Mandatory: compute the relaxed (properly)

- Add `opt(modredundant,tight)` to input (well-optimized geometries improve reproducibility)
- In case of DFT, add `int(grid=ultrafine)` to improve accuracy.
- In case of G03, add `scf(tight)`, to improve accuracy. This is the default in G09.
- Add `NoSymm` to avoid locking into undesired transition states

Recommended: limit the scan symmetry-inequivalent points

- For example, in the case of ethane, scan the H-C-C-H angle from 0 to 60 degrees.
- Rotational symmetry and parity (even or odd) can be imposed in TAMkin.

Recommended: organize your files

- A separate directory for every Gaussian job
- Use the same name for each type of file: `gaussian.com gaussian.log gaussian.chk guassian.fchk`. This facilitates automation.

Common pitfalls when using Gaussian

Gaussian is by no means perfect.

- Geometry Convergence failures are common in complex systems.
  → Restart optimization with best geometry so far (g-chkopt.py)
- Optimization to a transition state is typically harder. Common problems include: (i) failure to converge at all, (ii) convergence to the wrong transition state (e.g. remote methyl group rotates by 60 deg)
  1. Good initial guess of geometry. (from different LOT)
  2. Constrain irrelevant degrees of freedom.
  3. Use `opt(CalcFC)` or `opt(CalcAll)` in input. (expensive!)
     (poor initial guess? → add `noeigentest` to `opt` options)
  4. Some TSs are ill-define with a single reference method
- Cryptic error messages: Google.

Overview of examples

Ethanol

- 01: Normal mode analysis: vanilla
- 02: Normal mode analysis: fixed external degrees of freedom
- 03: Normal mode analysis: PHVA

Ethane

- 04: Single partition function
- 05: Single partition function with free rotor
- 06: Single partition function with hindered rotor (Gaussian scan)
- 07: Single partition function with hindered rotor (model potential)
- 08: Single partition function at pressure ≠ 1 atm

Ethene Ethyl addition

- 09: Chemical equilibrium
- 10: Kinetic parameters
- 11: Tunneling

A la carte ...