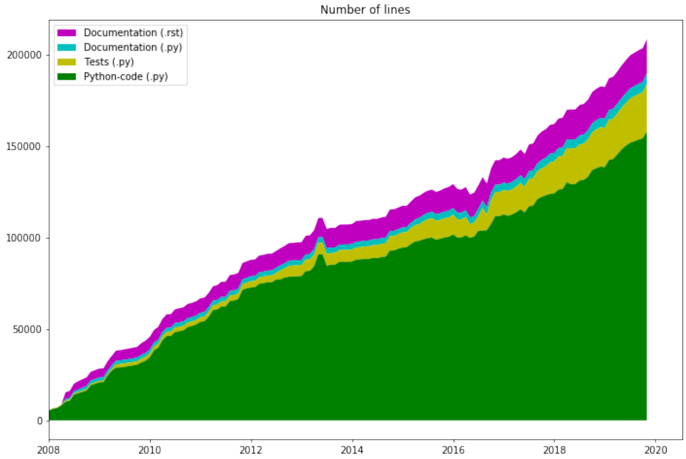
ASE: Plans and developments

Ask Hjorth Larsen

Simune Atomistics S.L.
Avenida de Tolosa 76, Donostia-San Sebastián, Spain

November 21, 2019



(Thanks @jensj)

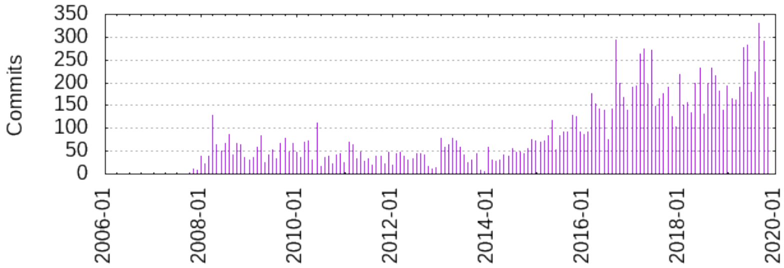


Figure: Commits per month

Development workflow

- Submit merge request on Gitlab
- Submitter fights desperately with CI tests
- Someone reviews the code
- (Right now someone == me, usually)
- We need experts for reviewing changes to exotic parts!

Common review comments

- Remember to write test
- Remember to add documentation
- Name the variables descriptively
- Don't repeat code, write a function
- Use existing implementation of X

Project growth: Diverging use cases

Historical "evolutionary" growth

- Informal, "interactive" workflow
- Focus on typical use cases (e.g. implicit I/O, parallelisation)
- Not so important if calculators work the same, user can adapt

Improve: Programmatic use and high-throughput

- Simple object states
- High level of generality (at expense of convenience)
- No magic shortcuts and assumptions
- Stable programming interface
- Limit scope and program size somehow (for libraries)

“Code strain”

More features ⇒ Larger codebase ⇒ more central maintenance
Diverging use cases lead to “code strain”:

- Add C code for superspeed? But all users will need to compile.
- Add dependencies? Often inconvenient on supercomputers.
- Add parameters for convenient workflow? API interface bloat.

We are mixing “user classes” and “programming interface” classes.
Possible solution: Modularize, split into multiple libraries.

A word on funding

- Thanks to Simune Atomistics for supporting ASE
- Open source-friendly company

Introducing ASAP

- (Not related to ASE calculator of the same name)
- Integrated program to perform atomistic simulations
- Currently under development
- Currently focusing on Siesta
- (However: GUI has proprietary license)

Band structures in 2016 (from gpaw 1.1.0 docs)

```
from ase.dft.kpoints import ibz_points, get_bandpath
points = ibz_points['fcc']
G = points['Gamma']
X = points['X']
W = points['W']
K = points['K']
L = points['L']
kpts, x, X = get_bandpath([W, L, G, X, W, K], calc.atoms.ce
calc.set(kpts=kpts)
atoms.get_potential_energy()
e_kn = np.array([calc.get_eigenvalues(k) for k in range(len
```

- ASE knows the special points, builds band path
- But: Many imports, repetitive work
- Code does not even entirely fit on a slide

Band structures in 2017

After “spring cleaning” for ASE paper:

```
calc.set(kpts={'path': 'GXWKL', 'npoints': 60})
atoms.get_potential_energy()
bs = calc.band_structure()
bs.plot(filename='bandstructure.png', show=True, emax=10.0)
```

- Pass special points as convenient string
- ASE automatically recognizes unit cell
- Shortcut on calculator for extracting band structure
- But: Logic hidden beneath deep layers
- How do I see where `'X'` is without calculating?

Working with band structures now

```
>>> print(atoms.cell.get_bravais_lattice().description())
FCC(a=5.43)
  Variant name: FCC
  Special point names: GKLUWX
  Default path: GXWKGLUWLK,UX

  Special point coordinates:
    G   0.0000  0.0000  0.0000
    K   0.3750  0.3750  0.7500
    L   0.5000  0.5000  0.5000
    U   0.6250  0.2500  0.6250
    W   0.5000  0.2500  0.7500
    X   0.5000  0.0000  0.5000

>>> print(atoms.cell.bandpath('GXWKL'))
BandPath(path='GXWKL', cell=[3x3], special_points={GKLUWX},
>>> bs = calculate_band_structure(atoms, path)
>>> bs.write('bs.json')
```

Simple object designs

Push for simpler objects in ASE

- Small objects with well-defined data and limited roles
- Meaningful string representations
- Easy plotting: `obj.plot(...)`
- Easy I/O: `obj.write('bs.json')`
- Command-line integration: `$ ase band-structure bs.json`
- In short: We always "know what we have" in front of us.

Complex objects

- Most objects in ASE are not simple
- Different levels and types of information stored
- How can you tell whether two objects are equal?
- Can you save to file and completely restore?
- Sequential coupling

- Sequential coupling: Workflow becomes "magic incantation"
- Must call methods in right order:

```
obj = MyClass(...)
obj.initialize()
obj.calculate()
obj.read()
x = obj.useful_method()
```

- Complex state: Not clear what the object can do and when.

Source: Francisco Goya / Wikipedia

Solution: Split into two or more objects:

```
calculator = ThingCalculator(...)
result = calculator.calculate()
result.plot()
```

- Simple state: Both objects are fully operational
- Portable outputs! Possible to represent outputs from another program in common framework.
- Currently: Always rely on ASE implementation

Proposal

- Calculator → Calculator + CalculatorOutput
- Vibrations → VibrationCalculator + VibrationalModes
- Phonons → PhononCalculator + PhononModes

Also clean up others (EOS, NEB, . . .).

Complexity of Atoms and calculators

Having calculators

- Do Atoms have a calculator? Maybe.
- Does the calculator have results on it? Maybe.
- Can the calculator actually calculate? Maybe.

In other words: If you have an Atoms object, you don't entirely know what you really have.

Roles of calculators in ASE

- Store input parameters
- Write input file / send inputs
- Calculate (run external code / call big function)
- Read/load outputs
- Store outputs
- Caching, cache invalidation
- “Restart” (whatever that means)
- Manage “label”

Calculators are difficult to write

- New contributors rarely understand how to write calculators, leading to redundant implementations
- Difficult to understand interactions with `Calculator` superclass
- State is too complex

Idea 1: "Data-driven" calculator

- Identify everything that any calculator ever wants to do
- Write a few actual classes to support those things
- Express every calculator as static information on which those calculators run
- Advantage: Minimizes code which we cannot easily test

Snippet from Quantum Espresso calculator:

```
def set(self, **kwargs):
    changed_parameters = FileIOCalculator.set(self, **k
    if changed_parameters:
        self.reset()

def write_input(self, atoms, properties=None, system_ch
    FileIOCalculator.write_input(self, atoms, propertie
    io.write(self.label + '.pwi', atoms, **self.paramet

def read_results(self):
    output = io.read(self.label + '.pwo')
    self.calc = output.calc
    self.results = output.calc.results
```

Small methods forward information to specialized functions.
Espresso is the smallest and therefore the best calculator.

Tabulate static information in "calculator template":

```
class CalculatorTemplate:
    def __init__(self, name, implemented_properties,
                 command,
                 input_file, input_format,
                 output_file, output_format):
        <save variables on self>

    def new(self, **kwargs):
        calc = DataDrivenCalculator(template=self,
                                     **kwargs)
        return calc
```

- Single implementation: Guaranteed consistency, easy testing
- But: Some calculators automatically search for pseudopotentials
- Also: Some calculators use multiple input/output files

Idea 2: Split calculator class

- Calculator inputs: Name + parameters (dictionary)
- Engine: Execute actual calculation
- Loader: Loads outputs from files/memory
- Calculator outputs: Static buffer with output

How much can atoms/calculators be simplified?

Current object structure

- Atoms
- Atoms with optional calculator
- Atoms with optional calculator with optional results

Alternative object structure

- Atoms
- Calculator with Atoms
- Results object with calculator inputs and Atoms

Alternative structure is cleaner: Nothing is optional or uncertain.
Uncertain whether we can do this in practice though.

Conclusions

- It would be great to simplify many classes in ASE.
- Your help is appreciated!
- Can things be done compatibly or would we need an ase-4.0?