Development of a real space grid based
PAW-DFT Python code

Jens Jørgen Mortensen,
Lars B. Hansen,
Karsten W. Jacobsen,
Jens K. Nørskov
Outline of talk

- Motivation
- Algorithm
- Projector Augmented Wave method
- Double grid technique
- Results
- Python
- Conclusions
Motivation for using real space grids:

- Simple: Only one parameter (grid spacing)
- Flexible boundary conditions
  1) Cluster
  2) Wire
  3) Surface
  4) Bulk
- First step towards order-$N$
- No Fourier transforms
Grids

Finite Difference operators:

\[ h^2 \nabla^2 = \begin{pmatrix} 1 & -4 & 1 \\ 1 & -4 & 1 \\ 1 & -4 & 1 \end{pmatrix} \]

Parallelization:
Algorithm:

- LCAO
- Update density
- Symmetrize
- Calculate potentials
- Apply Hamiltonian
- Diagonalize subspace
- Find occupation numbers
- Calculate residuals
- Update wavefunctions
- Orthogonalize

Pulay mixing:
$$\tilde{n}_{i+1} = \sum_i \alpha_i \tilde{n}^{out}_i$$
minimize: $$\sum_i \alpha_i (\tilde{n}^{in}_i - \tilde{n}^{out}_i)$$

Multigrid Poisson solver using Mehrstellen operator

$$R = \hat{H} \Psi - \epsilon \Psi$$

$$\Psi' = \Psi + \lambda \hat{P} R$$
minimize: $$R' = \hat{H} \Psi' - \epsilon \Psi'$$

$$\Psi \leftarrow \Psi' + \lambda \hat{P} R'$$

Preconditioning ($\hat{P}$): one V-cycle with $$\hat{H} = -\frac{1}{2} \nabla^2$$

Projector Augmented-Wave method


Why PAW?
1) Exact all-electron formalism
2) Soft wavefunctions (like USPP)

Approximations:
1) Finite number of projectors
2) Truncated angular momentum expansions
3) Frozen core

Projectors
\[ \langle \tilde{p}_i^a | \tilde{\Phi}_i^a \rangle = \delta_{ij} \]
\[ \Psi = \sum_a \sum_{nlm} (\phi_{nlm}^a - \tilde{\phi}_{nlm}^a) \langle \tilde{p}^a_{nlm} | \Psi \rangle + \Psi \]

All-electron
\[ \tilde{\rho} = \sum_a \Psi^2 + \sum_a \sum_{lm} Q_{lm}^a \tilde{g}_{lm}(r - R^a) \]
Double grid technique:

Projection: \( \langle \widetilde{\Psi}_n | \widetilde{p}_i^a \rangle \)

Interpolation: \( \widetilde{\Psi}(r) = \sum_R I_{rR} \overline{\Psi}(R) \)

\[
\langle \widetilde{\Psi} | \widetilde{p} \rangle = \sum_r \overline{\Psi}(r) \overline{p}(r) \Delta v = \sum_r \sum_R I_{rR} \overline{\Psi}(R) \overline{p}(r) \Delta v = \sum_R \overline{\Psi}(R) \sum_r I_{rR} \overline{p}(r) \frac{\Delta v}{\overline{p}(R)} \Delta V
\]

# Test of accuracy

## Nitrogen molecule:

<table>
<thead>
<tr>
<th>XC</th>
<th>PAW</th>
<th>d (bohr)</th>
<th>E (eV)</th>
<th>PAW</th>
<th>E (eV)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>all-electron</td>
<td>Dacapo</td>
<td>PAW</td>
<td>all-electron</td>
<td>Dacapo</td>
</tr>
<tr>
<td>LDA</td>
<td>2.072</td>
<td>2.071</td>
<td>11.59</td>
<td>11.58</td>
<td>10.54</td>
</tr>
<tr>
<td>PBE</td>
<td>2.086</td>
<td>2.084</td>
<td>2.116</td>
<td>10.55</td>
<td>10.05</td>
</tr>
<tr>
<td>revPBE</td>
<td>2.091</td>
<td>2.089</td>
<td>10.06</td>
<td>10.05</td>
<td>9.50</td>
</tr>
</tbody>
</table>

## Bulk aluminium:

<table>
<thead>
<tr>
<th>structure</th>
<th>a (Å)</th>
<th>B (GPa)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PAW</td>
<td>all-electron</td>
</tr>
<tr>
<td>FCC</td>
<td>3.993</td>
<td>3.983</td>
</tr>
<tr>
<td>BCC</td>
<td>3.201</td>
<td>3.193</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>formation energy</th>
<th>ΔE (eV)</th>
<th>PaW</th>
<th>all-electron</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCC – FCC</td>
<td>0.10</td>
<td>0.08</td>
<td></td>
</tr>
</tbody>
</table>
N$_2$ molecule

Convergence:

\[ \Delta E \sim h^4 \]

Grid noise:

\[ h = 0.2 \text{ Å} \]
Test: \( \text{H}_2 \) dissociation on Al(110)

unit cell: 5 layers, (2x1) cell
k-points: (6x2) IBZ: 3
\( h = 0.23 \, \text{Å} \)
Why use Python?

Premature optimization is the root of all evil.

Tony Hoare

By choosing a lower level language, like C++, at the start of your project, rather than a higher level one, like Python, you ARE optimizing WAY prematurely.

Alex Martelli
The ultimate goal:

Python: 5600 lines
C++: 1800 lines

- restrictions
- interpolations
- symmetrization
- finite difference Laplacian
- finite difference gradients
- exchange-correlation functionals

BLAS, LAPACK, Numeric
Python is Fast!

1) No compilation and linking
2) No declarations and .h files
3) Easy interactive debugging
4) No memory management
5) Many bugs don't exist in Python
6) Object Oriented
Future

• d-projectors
• Non-spherical
  – compensation charges
  – exchange-correlation energy
• Scalar-relativistic data sets
• Parallelization
• ...
• Order-N
Conclusions

• It is possible to do PAW calculations efficiently in real space.

• The Python/C++ combination works well for this type of work.
How fast is Python?

The number $e$:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

# include `<iostream>`

```cpp
int main()
{
    double e = 0.0;
    double f = 1.0;
    for (int n = 0; n < 100; n++)
    {
        e += f;
        f /= n + 1;
    }
    cout << "e = " << e << endl;
}
```

300 µs

```python
e = 0.0
f = 1.0
for n in range(100):
    e += f
    f /= n + 1
print "e =", e
```

6 µs

```python
print "e =", 1 + sum(divide.accumulate(arange(1, 100, typecode=Float)))
```