Automatic Differentiation for Quantum Electronic Structure

Differentiable Density Functional Theory in DFTK.jl

Markus Towara[1], Niklas Schmitz[2], Gaspard Kemlin[3]

JuliaCon 2022

[1] RWTH Aachen University
[2] TU Berlin
[3] École des Ponts ParisTech & Inria Paris

Team Intro

Markus Towara
PostDoc @ RWTH Aachen University. Background in AD and numerical simulation (Finite Volume CFD, mainly C++).

Niklas Schmitz
MSc Student @ TU Berlin. Computer science and machine learning.

Gaspard Kemlin
PhD Candidate @ École des Ponts ParisTech & Inria Paris, team MATHERIALS. Background in applied mathematics and numerical analysis.

Joint work with M.F. Herbst and A. Levitt

Overview

Introduction to DFTK.jl and DFT

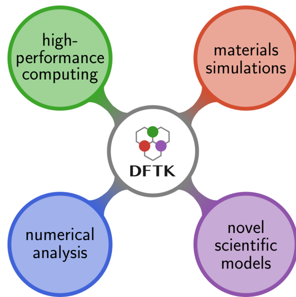
Foundations of AD

Application of AD to DFTK.jl

Demo

Introduction to DFTK.jl and DFT

Density-functional toolkit[1] – `https://dftk.org`



- `Julia` code for plane-wave DFT
- Fully composable with `Julia` ecosystem:
  - Arbitrary precision
  - Algorithmic Differentiation (AD)
  - Numerical error control

- Both suitable for mathematical developments and relevant applications
  - 1D problems, toy models for rigourous analysis
  - DFT > 800 electrons
- 3 years of development (M.F. Herbst and A. Levitt) and $\sim$ 7k lines of code

[1] M. F. Herbst, A. Levitt and E. Cancès. JuliaCon Proceedings, 3, 69 (2021).

Typical Workflow of DFT Simulations

1. Setup

- model
- atoms positions & types
- lattice
- basis

2. Solve

- compute self-consistent field
- obtain wave function

3. Postprocess

- energy
- forces
- stresses
- etc.

We want to backpropagate through all phases.

Density Functional Theory in one slide

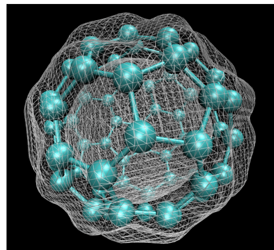
- Self-Consistent Field (SCF) procedure

$$0 = f(P_*, \lambda) = f_{\mathsf{FD}}(H^\lambda(P_*)) - P_* \quad \Leftrightarrow \quad P_* \in \operatorname{argmin} E^\lambda(P)$$

  - P : density matrix (describes the electronic states)
  - λ : parameters (model, atomic positions, electric field, ...)
  - f_{FD} : Fermi-Dirac function, for $(\varepsilon_n, \phi_n)_{n\in\mathbb{N}}$ eigenpairs of H

$$f_{\mathsf{FD}}(H) = \sum_{n\in\mathbb{N}} f_{\mathsf{FD}}(\varepsilon_n)|\phi_n\rangle\langle\phi_n|$$

  - H^λ: nonlinear Kohn-Sham Hamiltonian
  - E: energy



Isosurface of ground-state electron density of Fullerene, as calculated with DFT (source: Wikimedia Commons)

- Defines $P(\lambda)$ with implicit dependency on the parameters

Why are we so interested in derivatives ?

Most quantities of interest are computed as derivatives of another quantity of interest:

$$\frac{\mathrm{d}A(P)}{\mathrm{d}\lambda} = \frac{\partial A}{\partial \lambda} + \frac{\partial A}{\partial P}\frac{\partial P}{\partial \lambda}$$

- Forces: $A = E, \lambda = R$ (atomic positions)
- Stresses: $A = E, \lambda = \mathbb{L}$ (unit cell vectors)
- Polarizability: $A =$ dipole moment, $\lambda = \mathcal{E}$ (electric field)
- Sensitivity to any parameter (e.g. model parameters) for ML applications
- ...

Hellmann-Feynman theorem

$$\frac{\mathrm{d}A(P)}{\mathrm{d}\lambda} = \frac{\partial A}{\partial \lambda} + \frac{\partial A}{\partial P}\frac{\partial P}{\partial \lambda}$$

Special case of $A = E$:

- Recall $P_* \in$ argmin $E(P)$ $\quad \Rightarrow \quad \left.\frac{\partial E}{\partial P}\right|_* = 0$
- Hellmann-Feynman theorem

$$\left.\frac{\mathrm{d}E}{\mathrm{d}\lambda}\right|_* = \left.\frac{\partial E}{\partial \lambda}\right|_*$$

- First energy derivatives are (comparatively) easy!

Response theory

- If $A \neq E$ we need $\frac{\partial P}{\partial \lambda}$!
- Consider at $\lambda = \lambda_*$ and corresponding P_* and H_*:

$$
\begin{aligned}
0 &= \frac{\partial}{\partial \lambda} \left[f_{\mathsf{FD}}\Big(H^\lambda(P) \Big) - P \right] \Bigg|_* \\
&= f'_{\mathsf{FD}}(H_*) \cdot \frac{\partial H^\lambda}{\partial \lambda}\bigg|_* + \frac{\partial P}{\partial \lambda}\bigg|_* \cdot \frac{\partial}{\partial P} \left[f_{\mathsf{FD}}\Big(H^\lambda(P) \Big) - P \right] \bigg|_* \\
&= f'_{\mathsf{FD}}(H_*) \cdot \frac{\partial H^\lambda}{\partial \lambda}\bigg|_* + \frac{\partial P}{\partial \lambda}\bigg|_* \cdot \left[f'_{\mathsf{FD}}(H_*) \cdot \boldsymbol{K}(P_*) - I \right] \\
&= \boldsymbol{\chi}_0(H_*) \cdot \frac{\partial H^\lambda}{\partial \lambda}\bigg|_* - \frac{\partial P}{\partial \lambda}\bigg|_* \cdot \left[I - \boldsymbol{\chi}_0(H_*) \cdot \boldsymbol{K}(P_*) \right]
\end{aligned}
$$

where $\boldsymbol{K} = \dfrac{\partial H^{\lambda_*}}{\partial P}$ and $\boldsymbol{\chi}_0(H_*) = f'_{\mathsf{FD}}(H_*)$

Sterneheimer equation

Sternheimer equation:

$$\left.\frac{\partial P}{\partial \lambda}\right|_{*} = -\left[\boldsymbol{\Omega}(H_*) + \boldsymbol{K}(P_*)\right]^{-1} \left.\frac{\partial H^{\lambda}}{\partial \lambda}\right|_{*}$$

where $\boldsymbol{\Omega}(H_*) = -\left[\boldsymbol{\chi}_0(H_*)\right]^{-1}$.

$\boldsymbol{\Omega}(H_*) + \boldsymbol{K}(P_*)$ is self-adjoint $\Rightarrow$ good for both tangent *and* adjoint mode!

Foundations of AD

AD Basics[2]

- Assume $y = f(x)$ with $x \in \mathbb{R}^n, y \in \mathbb{R}^m$
- Forward (tangent) AD: $\dot{y} = \dot{f}(x, \dot{x}) = \nabla f \cdot \dot{x}$

  Get Jacobian at cost $O(n \cdot cost(f))$ by letting $\dot{x} \in \mathbb{R}^n$ range over e_i
- Reverse (adjoint) AD: $\bar{x} = \bar{f}(x, \bar{y}) = \bar{y} \cdot \nabla f$

  Get Jacobian at cost $O(m \cdot cost(f))$ by letting $\bar{y} \in \mathbb{R}^m$ range over e_i
- Often $m \ll n$ or even $m = 1$ (e.g. Least Squares sum of outputs)
- Modes can be recursively combined to obtain higher derivatives
- Sparsity in Jacobians / Hessians can be exploited by coloring approaches

[2] A. Griewank, A. Walther: Evaluating Derivatives, 2nd Edition

Chain Rule

- Suppose function $h(g(f(x)))$
- Then $\frac{\mathrm{d}h}{\mathrm{d}x} = \frac{\mathrm{d}h}{\mathrm{d}g} \cdot \frac{\mathrm{d}g}{\mathrm{d}f} \cdot \frac{\mathrm{d}f}{\mathrm{d}x}$
- Order in which product is evaluated can be crucial:
  - $\frac{\mathrm{d}h}{\mathrm{d}g} \cdot (\frac{\mathrm{d}g}{\mathrm{d}f} \cdot \frac{\mathrm{d}f}{\mathrm{d}x})$ forward (tangent) mode
  - $(\frac{\mathrm{d}h}{\mathrm{d}g} \cdot \frac{\mathrm{d}g}{\mathrm{d}f}) \cdot \frac{\mathrm{d}f}{\mathrm{d}x}$ adjoint (reverse) mode
- However: Reverse mode differentiation can (in general) not be performed alongside primal evaluation (split mode AD)
- We use `Zygote` for reverse AD and heavily rely on `ChainRules.jl` to specify custom derivatives for parts of the chain rule product

Custom Backpropagation Rules

Two main reasons to specify custom rules:

- Use analytical (domain) knowledge, e.g. for linear equation systems[3]:

$$x = A \setminus b \Rightarrow \bar{b} = A^T \setminus \bar{x}$$
$$\bar{A} = -x \cdot \bar{b}^T$$

- Working around issues with adjoint code generation:
  (e.g. mutation not supported by Zygote, calls into foreign code) which cannot feasibly be fixed in the primal codebase (for e.g. performance reasons).

[3] M. Giles: *Collected Matrix Derivative Results for Forward and Reverse Mode Algorithmic Differentiation*, 2008

ChainRules Example for Linear Equations

```julia
solve(A, b) = A \ b

function ChainRulesCore.rrule(::typeof(solve), A, b)
    x = A \ b
    function solve_pullback(∂x)
        ∂b = A' \ ∂x
        ∂A = -x * ∂b'
        return ChainRulesCore.NoTangent(), ∂A, ∂b
    end
    return x, solve_pullback
end

Zygote.gradient((A,b) -> sum(abs2, solve(A, b)), A, b)
```

- Custom ChainRules rrule returns primal result, as well as a callback which will be called during backpropagation
- Primal can be left unchanged or rewritten
- Pullback can be written explicitly or generated by AD (e.g. from altered primal)

Application of AD to DFTK.jl

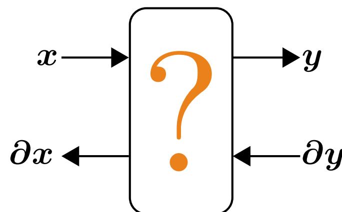
Common AD Rule Patterns

The classic

- handwritten derivative

but also

- linear functions (are their own derivative)
- alternative primal (callback into AD)
- implicit differentiation



Linear Rules

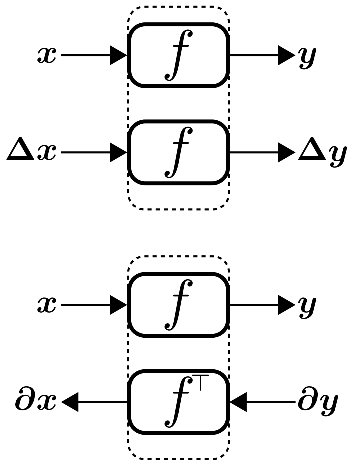
Linear frules

"This function is linear and thus its own derivative. Apply it on forward tangents."

Linear rrules

"This function is linear and thus its own derivative. Apply its transpose on reverse cotangents."

Example f: Fast Fourier Transform



Example: Custom FFT Rules

```julia
using ChainRulesCore

function ChainRulesCore.rrule(::typeof(r_to_G), basis::PlaneWaveBasis, f_real::AbstractArray)
    f_fourier = r_to_G(basis, f_real)
    function r_to_G_pullback(∂f_fourier)
        ∂f_real = G_to_r(basis, complex(∂f_fourier)) * basis.r_to_G_normalization / basis.G_to_r_normalization
        ∂normalization = real(dot(∂f_fourier, f_fourier)) / basis.r_to_G_normalization
        ∂basis = Tangent{typeof(basis)}(;r_to_G_normalization=∂normalization)
        return NoTangent(), ∂basis, real(∂f_real)
    end
    return f_fourier, r_to_G_pullback
end

function ChainRulesCore.rrule(::typeof(G_to_r), basis::PlaneWaveBasis, f_fourier::AbstractArray; kwargs...)
    f_real = G_to_r(basis, f_fourier; kwargs...)
    function G_to_r_pullback(∂f_real)
        ∂f_fourier = r_to_G(basis, real(∂f_real)) * basis.G_to_r_normalization / basis.r_to_G_normalization
        ∂normalization = real(dot(∂f_real, f_real)) / basis.G_to_r_normalization
        ∂basis = Tangent{typeof(basis)}(;G_to_r_normalization=∂normalization)
        return NoTangent(), ∂basis, ∂f_fourier
    end
    return f_real, G_to_r_pullback
end
```

Figure 1: Reverse-mode rules leveraging FFT duality

Example: Custom FFT Rules

```julia
using ChainRulesCore

function ChainRulesCore.rrule(::typeof(r_to_G), basis::PlaneWaveBasis, f_real::AbstractArray)
    f_fourier = r_to_G(basis, f_real)
    function r_to_G_pullback(∂f_fourier)
        ∂f_real = G_to_r(basis, complex(∂f_fourier)) * basis.r_to_G_normalization / basis.G_to_r_normalization
        ∂normalization = real(dot(∂f_fourier, f_fourier)) / basis.r_to_G_normalization
        ∂basis = Tangent{typeof(basis)}(;r_to_G_normalization=∂normalization)
        return NoTangent(), ∂basis, real(∂f_real)
    end
    return f_fourier, r_to_G_pullback
end

function ChainRulesCore.rrule(::typeof(G_to_r), basis::PlaneWaveBasis, f_fourier::AbstractArray; kwargs...)
    f_real = G_to_r(basis, f_fourier; kwargs...)
    function G_to_r_pullback(∂f_real)
        ∂f_fourier = r_to_G(basis, real(∂f_real)) * basis.G_to_r_normalization / basis.r_to_G_normalization
        ∂normalization = real(dot(∂f_real, f_real)) / basis.G_to_r_normalization
        ∂basis = Tangent{typeof(basis)}(;G_to_r_normalization=∂normalization)
        return NoTangent(), ∂basis, ∂f_fourier
    end
    return f_real, G_to_r_pullback
end
```

Figure 2: Reverse-mode rules leveraging FFT duality

Alternative Primal Rules

Alternative Primal

"To differentiate this complicated function, differentiate this simpler equivalent function."

Uses `ChainRules.jl` feature for calling back into AD[4].

- mostly for prototyping
- simplified is e.g. non-mutating
- can sidestep large non-differentiable auxiliary computations

---

[4] https://juliadiff.org/ChainRulesCore.jl/stable/rule_author/superpowers/ruleconfig.html

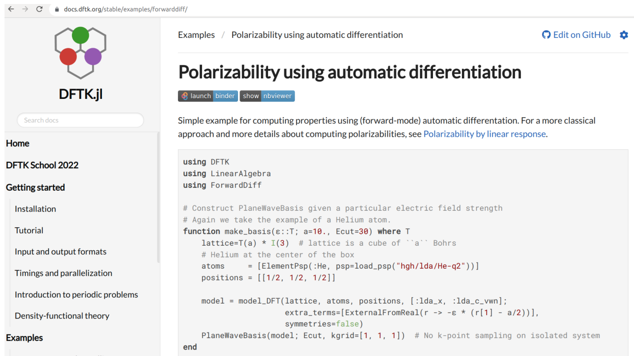
Implicit Differentiation: SCF rrule

```
function ChainRulesCore.rrule(
        config::RuleConfig{>:HasReverseMode},
        ::typeof(self_consistent_field),
        basis::PlaneWaveBasis{T};
        kwargs...) where {T}
    scfres = self_consistent_field(basis; kwargs...)
    function self_consistent_field_pullback(∂scfres)
        ...
        ∂basis = ...# specialized linear solve
        ...
        return NoTangent(), ∂basis
    end
    return scfres, self_consistent_field_pullback
end
```

Figure 3: The most central rrule: Differentiating the SCF solver.

Demo

Demo: Polarizability (Forward + Reverse Mode, FD)



https://docs.dftk.org/stable/examples/forwarddiff/

https://github.com/JuliaMolSim/DFTK.jl/blob/zygote-juliacon/examples/zygote_polarizability.jl

Summary & Outlook

Summary:

- Self-adjointness of the SCF is leveraged extensively
- Implicit differentiation ties well with ChainRules
- Writing performant code which is AD friendly is hard

Up next:

- Prototype different AD solutions (e.g. Enzyme)
- Publications for forward and reverse AD in progress

Acknowledgements

M.F. Herbst

A. Levitt

Funded under the Excellence Strategy of the Federal Government and the Länder