ODL

A Python framework for rapid prototyping in inverse problems

Jonas Adler[1, 2] Holger Kohr[3] Ozan Öktem[1]

[1] Department of Mathematics
KTH - Royal Institute of Technology, Stockholm

[2] Research and Physics
Elekta, Stockholm

[3] Thermo Fisher Scientific, Eindhoven

Why a new software framework?

Multiple modalities: CT, CBCT, PET, SPECT, spectral CT, phase contrast CT, electron tomography, MRI, HADAF-STEM …

Why a new software framework?

Multiple modalities: CT, CBCT, PET, SPECT, spectral CT, phase contrast CT, electron tomography, MRI, HADAF-STEM …

Collaborative research: Need to share implementations of common concepts

Why a new software framework?

Multiple modalities: CT, CBCT, PET, SPECT, spectral CT, phase contrast CT, electron tomography, MRI, HADAF-STEM …

Collaborative research: Need to share implementations of common concepts

Reproducible research: Not enough to share theory and pseudocode, also need to share data and concrete implementations
$\rightsquigarrow$ Software components need to be usable by others.

Why a new software framework?

Multiple modalities: CT, CBCT, PET, SPECT, spectral CT, phase contrast CT, electron tomography, MRI, HADAF-STEM ...

Collaborative research: Need to share implementations of common concepts

Reproducible research: Not enough to share theory and pseudocode, also need to share data and concrete implementations
$\rightsquigarrow$ Software components need to be usable by others.

Flexibility: Mathematical structures/notions *re-usable across modalities*
$\rightsquigarrow$ Make it easy to “play around” with new ideas and combine concepts.

Why a new software framework?

Multiple modalities: CT, CBCT, PET, SPECT, spectral CT, phase contrast CT, electron tomography, MRI, HADAF-STEM …

Collaborative research: Need to share implementations of common concepts

Reproducible research: Not enough to share theory and pseudocode, also need to share data and concrete implementations
$\rightsquigarrow$ Software components need to be usable by others.

Flexibility: Mathematical structures/notions *re-usable across modalities*
$\rightsquigarrow$ Make it easy to “play around” with new ideas and combine concepts.

Conclusion: Need a *common software framework* to exchange implementations of concepts and methods.

Why a new software framework?

Requirements on a software framework:

Allow formulation and solution of inverse problems in a *common language*.

Why a new software framework?

Requirements on a software framework:

- Allow formulation and solution of inverse problems in a *common language*.
- Make implementations *re-usable* and *extendable*.

Why a new software framework?

Requirements on a software framework:

- Allow formulation and solution of inverse problems in a *common language*.
- Make implementations *re-usable* and *extendable*.
- Enable *fast prototyping* on *clinically relevant data*.

Why a new software framework?

Requirements on a software framework:

- Allow formulation and solution of inverse problems in a *common language*.
- Make implementations *re-usable* and *extendable*.
- Enable *fast prototyping* on *clinically relevant data*.
- Leverage the power of *existing libraries*.

Why a new software framework?

Requirements on a software framework:

- Allow formulation and solution of inverse problems in a *common language*.
- Make implementations *re-usable* and *extendable*.
- Enable *fast prototyping* on *clinically relevant data*.
- Leverage the power of *existing libraries*.

Initial situation: No existing framework fit our purpose.

Operator Discretization Library

Main components:

Functional analysis module

Handling of *vector spaces*, *operators*, *discretizations* – generally with a *continuous* point of view

Operator Discretization Library

Main components:

Functional analysis module

Handling of *vector spaces*, *operators*, *discretizations* – generally with a *continuous* point of view

Optimization methods module

General-purpose optimization methods suitable for solving inverse problems.

Operator Discretization Library

Main components:

Functional analysis module

Handling of *vector spaces*, *operators*, *discretizations* – generally with a *continuous* point of view

Optimization methods module

General-purpose optimization methods suitable for solving inverse problems.

Tomography module

Acquisition *geometries* and *forward operators* for tomographic applications.

Operator Discretization Library

Main components:

Library of atomic mathematical components

– Deformation operators
– Function transforms: wavelet, Fourier, shearlet, ...
– Differential operators: partial derivative, gradient, Laplacian, ...
– Discretization-related: (re-)sampling, interpolation, domain extension, ...

Operator Discretization Library

Main components:

Library of atomic mathematical components

– Deformation operators
– Function transforms: wavelet, Fourier, shearlet, …
– Differential operators: partial derivative, gradient, Laplacian, …
– Discretization-related: (re-)sampling, interpolation, domain extension, …

Utility functions

– Visualization: Slice viewer, real time plotting, …
– Phantoms: Shepp-Logan, FORBILD, Defrise, …
– Data I/O: MRC2014, Mayo Clinic, …

Operator Discretization Library

Main components:

User-contributed modules

“Fast track” for experimental or slightly exotic code

– Figures of Merit (FOMs) for image quality assessment

– Handlers for specific data formats or geometries

– Functionality to download and import public datasets

– Wrappers for Deep Learning frameworks: Tensorflow, Theano, Pytorch, …

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \, \mathrm{TV}(f) \right]$$

Components:

Reconstruction space X

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \, \mathrm{TV}(f) \right]$$

Components:

- Reconstruction space X
- Data space Y

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \, \mathrm{TV}(f) \right]$$

Components:

- Reconstruction space X
- Data space Y
- Forward operator $\mathcal{T} : X \to Y$

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \, \mathrm{TV}(f) \right]$$

Components:

- Reconstruction space X
- Data space Y
- Forward operator $\mathcal{T} \colon X \to Y$
- Data $g \in Y$

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \, \mathrm{TV}(f) \right]$$

Components:

- Reconstruction space X
- Data space Y
- Forward operator $\mathcal{T} : X \to Y$
- Data $g \in Y$

Data discrepancy functional $\| \cdot - g\|_Y^2$

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \, \mathrm{TV}(f) \right]$$

Components:

- Reconstruction space X
- Data space Y
- Forward operator $\mathcal{T} : X \to Y$
- Data $g \in Y$
- Data discrepancy functional $\| \cdot - g\|_Y^2$
- Regularization parameter $\lambda > 0$

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \, \mathrm{TV}(f) \right]$$

Components:

- Reconstruction space X
- Data space Y
- Forward operator $\mathcal{T} \colon X \to Y$
- Data $g \in Y$
- Data discrepancy functional $\| \cdot - g\|_Y^2$
- Regularization parameter $\lambda > 0$
- Regularization functional $\mathrm{TV}(\,\cdot\,)$

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \, \mathrm{TV}(f) \right]$$

Components:

- Reconstruction space X
- Data space Y
- Forward operator $\mathcal{T} \colon X \to Y$
- Data $g \in Y$
- Data discrepancy functional $\| \cdot - g\|_Y^2$
- Regularization parameter $\lambda > 0$
- Regularization functional $\mathrm{TV}(\,\cdot\,)$

$\rightsquigarrow$ (almost) freely exchangeable "modules" in the mathematical formulation

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \, \mathrm{TV}(f) \right]$$

Components:

Reconstruction space X

Data space Y

Forward operator $\mathcal{T} \colon X \to Y$

Data $g \in Y$

Data discrepancy functional $\| \cdot - g\|_Y^2$

Regularization parameter $\lambda > 0$

Regularization functional $\mathrm{TV}(\,\cdot\,)$

$\rightsquigarrow$ (almost) freely exchangeable “modules” in the mathematical formulation

$\rightsquigarrow$ ODL maps them to software objects as closely as possible

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \operatorname{TV}(f) \right]$$

Components:

- Reconstruction space X
- Data space Y
- Forward operator $\mathcal{T} \colon X \to Y$
- Data $g \in Y$
- Data discrepancy functional $\| \cdot - g\|_Y^2$
- Regularization parameter $\lambda > 0$
- Regularization functional $\operatorname{TV}(\,\cdot\,)$

$\rightsquigarrow$ (almost) freely exchangeable "modules" in the mathematical formulation

$\rightsquigarrow$ ODL maps them to software objects as closely as possible

$\rightsquigarrow$ Mathematics as strong guideline for software design

Design principle: modularity

Consider a TV minimization problem

$$\min_{f \in X} \left[\|\mathcal{T}(f) - g\|_Y^2 + \lambda \, \mathrm{TV}(f) \right]$$

Components:

Reconstruction space X

Data space Y

Forward operator $\mathcal{T} \colon X \to Y$

Data $g \in Y$

Data discrepancy functional $\| \cdot - g\|_Y^2$

Regularization parameter $\lambda > 0$

Regularization functional $\mathrm{TV}(\,\cdot\,)$

- $\rightsquigarrow$ (almost) freely exchangeable “modules” in the mathematical formulation
- $\rightsquigarrow$ ODL maps them to software objects as closely as possible
- $\rightsquigarrow$ Mathematics as strong guideline for software design
- $\rightsquigarrow$ Makes the software “feel” natural

Design principle: abstraction

Landweber's method: Determine f from given data $g = \mathcal{T}(f)$ and initial guess f_0 by

$$f_{k+1} = f_k + \omega[\partial\mathcal{T}(f_k)]^*\big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \dots, K-1$$

Design principle: abstraction

Landweber's method: Determine f from given data $g = \mathcal{T}(f)$ and initial guess f_0 by

$$f_{k+1} = f_k + \omega [\partial \mathcal{T}(f_k)]^* \big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

```python
def landweber(T, f, g, omega, K):
    for i in range(K):
        f += omega * T.derivative(f).adjoint(g - T(f))
```

Design principle: abstraction

Landweber's method: Determine f from given data $g = \mathcal{T}(f)$ and initial guess f_0 by

$$f_{k+1} = f_k + \omega[\partial\mathcal{T}(f_k)]^*\big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

```python
def landweber(T, f, g, omega, K):
    for i in range(K):
        f += omega * T.derivative(f).adjoint(g - T(f))
```

Completely generic (expects operator, data, plus some parameters)

Design principle: abstraction

Landweber's method: Determine f from given data $g = \mathcal{T}(f)$ and initial guess f_0 by

$$f_{k+1} = f_k + \omega[\partial\mathcal{T}(f_k)]^*\big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

```python
def landweber(T, f, g, omega, K):
    for i in range(K):
        f += omega * T.derivative(f).adjoint(g - T(f))
```

Completely generic (expects operator, data, plus some parameters)

Uses abstract properties of operators in the iteration:

$\rightsquigarrow$ `T(f)` $\longleftrightarrow \mathcal{T}(f)$ (operator evaluation)

$\rightsquigarrow$ `T.derivative(f)` $\longleftrightarrow \partial\mathcal{T}(f)$ (derivative *operator* at f)

$\rightsquigarrow$ `T.derivative(f).adjoint` $\longleftrightarrow [\partial\mathcal{T}(f)]^*$ (adjoint of the derivative at f)

Design principle: abstraction

Landweber's method: Determine f from given data $g = \mathcal{T}(f)$ and initial guess f_0 by

$$f_{k+1} = f_k + \omega[\partial\mathcal{T}(f_k)]^*\big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

```
def landweber(T, f, g, omega, K):
    for i in range(K):
        f += omega * T.derivative(f).adjoint(g - T(f))
```

`T` is an `Operator` that implements a *generic, abstract* interface:
`domain`, `range`, `derivative`, `adjoint`, operator *evaluation*

Design principle: abstraction

Landweber's method: Determine f from given data $g = \mathcal{T}(f)$ and initial guess f_0 by

$$f_{k+1} = f_k + \omega[\partial\mathcal{T}(f_k)]^*\big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

```python
def landweber(T, f, g, omega, K):
    for i in range(K):
        f += omega * T.derivative(f).adjoint(g - T(f))
```

`T` is an `Operator` that implements a *generic, abstract* interface:
`domain`, `range`, `derivative`, `adjoint`, operator *evaluation*

Lots of tools to build complex operators from simple ones:
operator arithmetic `T + S`, composition `T * S`, product space operators etc.

Design principle: abstraction

Landweber's method: Determine f from given data $g = \mathcal{T}(f)$ and initial guess f_0 by

$$f_{k+1} = f_k + \omega[\partial\mathcal{T}(f_k)]^*\big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

```
def landweber(T, f, g, omega, K):
    for i in range(K):
        f += omega * T.derivative(f).adjoint(g - T(f))
```

`T` is an `Operator` that implements a *generic, abstract* interface: `domain`, `range`, `derivative`, `adjoint`, operator *evaluation*

Lots of tools to build complex operators from simple ones: operator arithmetic `T + S`, composition `T * S`, product space operators etc.

There are *many* readily implement operators in ODL, all implementing the above interface

Design principle: compartmentalization

Separates the “what” (abstract interface) of an object class from the “how” (concrete implementation)

Example: Fourier transform using `NumPy` FFT vs. `pyFFTW` vs. `cuFFT`

Design principle: compartmentalization

Separates the “what” (abstract interface) of an object class from the “how” (concrete implementation)
Example: Fourier transform using `NumPy` FFT vs. `pyFFTW` vs. `cuFFT`

Allows building generic APIs with the possibility for a new implementation in the future (*extensibility*)
Example: `L1Norm` as a concrete realization of the abstract `Functional`

Design principle: compartmentalization

- Separates the "what" (abstract interface) of an object class from the "how" (concrete implementation)
  Example: Fourier transform using `NumPy` FFT vs. `pyFFTW` vs. `cuFFT`
- Allows building generic APIs with the possibility for a new implementation in the future (*extensibility*)
  Example: `L1Norm` as a concrete realization of the abstract `Functional`
- Makes functions and classes individually testable

Design principle: compartmentalization

- Separates the "what" (abstract interface) of an object class from the "how" (concrete implementation)
  Example: Fourier transform using `NumPy` FFT vs. `pyFFTW` vs. `cuFFT`
- Allows building generic APIs with the possibility for a new implementation in the future (*extensibility*)
  Example: `L1Norm` as a concrete realization of the abstract `Functional`
- Makes functions and classes individually testable
- Documentation is bundled with the object and immediately visible to the user

Design principle: compartmentalization

- Separates the "what" (abstract interface) of an object class from the "how" (concrete implementation)
  Example: Fourier transform using `NumPy` FFT vs. `pyFFTW` vs. `cuFFT`
- Allows building generic APIs with the possibility for a new implementation in the future (*extensibility*)
  Example: `L1Norm` as a concrete realization of the abstract `Functional`
- Makes functions and classes individually testable
- Documentation is bundled with the object and immediately visible to the user
- Code becomes more maintainable, low risk for "god objects" or scope glide

Further design considerations

ODL is a *prototyping* framework, not a black-box solution

$\rightsquigarrow$ Give users freedom to experiment and tinker, do "unorthodox" things

$\rightsquigarrow$ Very little "intelligence" that guesses what a user wants

$\rightsquigarrow$ Instead: make things "just work" that a typical user would expect to work

Further design considerations

ODL is a *prototyping* framework, not a black-box solution

$\rightsquigarrow$ Give users freedom to experiment and tinker, do “unorthodox” things

$\rightsquigarrow$ Very little “intelligence” that guesses what a user wants

$\rightsquigarrow$ Instead: make things “just work” that a typical user would expect to work

It should be *fun* to explore the “What if?” scenarios in existing examples

Further design considerations

ODL is a *prototyping* framework, not a black-box solution

$\rightsquigarrow$ Give users freedom to experiment and tinker, do "unorthodox" things

$\rightsquigarrow$ Very little "intelligence" that guesses what a user wants

$\rightsquigarrow$ Instead: make things "just work" that a typical user would expect to work

It should be *fun* to explore the "What if?" scenarios in existing examples

Make use of external highly optimized code for heavy tasks if adequate

Further design considerations

ODL is a *prototyping* framework, not a black-box solution
$\leadsto$ Give users freedom to experiment and tinker, do “unorthodox” things
$\leadsto$ Very little “intelligence” that guesses what a user wants
$\leadsto$ Instead: make things “just work” that a typical user would expect to work

It should be *fun* to explore the “What if?” scenarios in existing examples

Make use of external highly optimized code for heavy tasks if adequate

Don’t sacrifice performance!
$\leadsto$ Use libraries in the most efficient way possible (avoid copies, operate in-place, work with low-dimensional arrays, vectorization, broadcasting, …)
$\leadsto$ Compute on the GPU whenever possible (new fast back-end coming soon)

Example: Tomography

Inverse Problem: Determine attenuation coefficient $\mu\colon \Omega \to \mathbb{R}$ from its ray transform $\mathcal{P}\colon L^2(\Omega) \to Y$ defined as

$$\mathcal{P}(\mu)(\ell) := \int_\ell \mu(x)\mathrm{d}x$$

for all lines ℓ.

Example: Tomography

Inverse Problem: Determine attenuation coefficient $\mu\colon \Omega \to \mathbb{R}$ from its ray transform $\mathcal{P}\colon L^2(\Omega) \to Y$ defined as

$$\mathcal{P}(\mu)(\ell) := \int_\ell \mu(x)\mathrm{d}x$$

for all lines ℓ.

Given: Noisy data

$$g(\ell) \approx \mathcal{P}(\mu)(\ell)$$

Example: Tomography

Inverse Problem: Determine attenuation coefficient $\mu\colon \Omega \to \mathbb{R}$ from its ray transform $\mathcal{P}\colon L^2(\Omega) \to Y$ defined as

$$\mathcal{P}(\mu)(\ell) := \int_\ell \mu(x)\mathrm{d}x$$

for all lines ℓ.

Given: Noisy data

$$g(\ell) \approx \mathcal{P}(\mu)(\ell)$$

Regularization: Conjugate gradient (CGLS) with early termination

Example: Tomography

Implementation steps:

Set up uniformly *discretized* image space $L^2(\Omega)$ with a rectangular domain Ω and $n_x \times n_y$ pixels

Example: Tomography

Implementation steps:

- Set up uniformly *discretized* image space $L^2(\Omega)$ with a rectangular domain Ω and $n_x \times n_y$ pixels
- Create parallel beam geometry with P angles and K detector pixels

Example: Tomography

Implementation steps:

- Set up uniformly *discretized* image space $L^2(\Omega)$ with a rectangular domain Ω and $n_x \times n_y$ pixels
- Create parallel beam geometry with P angles and K detector pixels
- Define ray transform $\mathcal{P} : L^2(\Omega) \to Y$ (the space Y is inferred from the geometry)

Example: Tomography

Implementation steps:

- Set up uniformly *discretized* image space $L^2(\Omega)$ with a rectangular domain Ω and $n_x \times n_y$ pixels
- Create parallel beam geometry with P angles and K detector pixels
- Define ray transform $\mathcal{P}: L^2(\Omega) \to Y$ (the space Y is inferred from the geometry)
- Solve inverse problem using CGLS

Example: Tomography

Implementation steps:

- Set up uniformly *discretized* image space $L^2(\Omega)$ with a rectangular domain Ω and $n_x \times n_y$ pixels
- Create parallel beam geometry with P angles and K detector pixels
- Define ray transform $\mathcal{P} \colon L^2(\Omega) \to Y$ (the space Y is inferred from the geometry)
- Solve inverse problem using CGLS
- Display the results

Example: Tomography

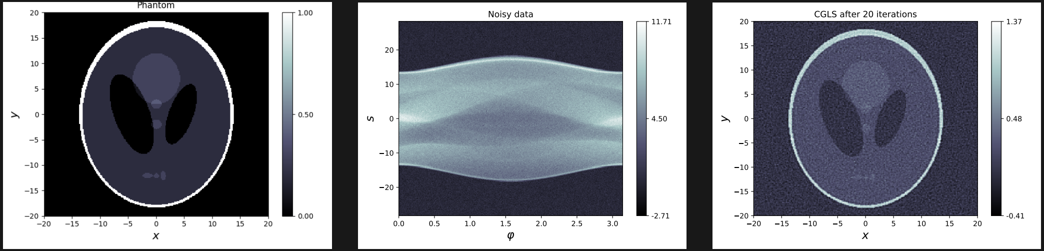
```python
# Create reconstruction space and ray transform
space = odl.uniform_discr([-20, -20], [20, 20], shape=(256, 256))
geometry = odl.tomo.parallel_beam_geometry(space, num_angles=1000)
ray_transform = odl.tomo.RayTransform(space, geometry)

# Create artificial data with around 5 % noise (data max = 10)
phantom = odl.phantom.shepp_logan(space, modified=True)
g = ray_transform(phantom)
g_noisy = g + 0.5 * odl.phantom.white_noise(ray_transform.range)

# Solve inverse problem
x = space.zero()
odl.solvers.conjugate_gradient_normal(ray_transform, x, g_noisy, niter=20)

# Display results
phantom.show('Phantom')
g_noisy.show('Noisy data')
x.show('CGLS after 20 iterations')
```

Example: Tomography



Conclusions and Outlook

Reproducible – scalable research requires rethinking scientific software

`github.com/odlgroup/odl`

Conclusions and Outlook

Reproducible – scalable research requires rethinking scientific software

ODL aims to solve these problems

`github.com/odlgroup/odl`

Conclusions and Outlook

Reproducible – scalable research requires rethinking scientific software

ODL aims to solve these problems

Well tested, used in $\approx$20 articles the last year

`github.com/odlgroup/odl`

Conclusions and Outlook

Reproducible – scalable research requires rethinking scientific software

ODL aims to solve these problems

Well tested, used in ≈20 articles the last year

Anyone is welcome to use and/or contribute!

`github.com/odlgroup/odl`