ODL – a Python framework for rapid prototyping in inverse problems

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

[1]Computational Imaging
Centrum Wiskunde & Informatica, Amsterdam

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

October 17, 2017
ODL training
Max IV, Lund

Why a new software framework?

- Multiple modalities: CT, CBCT, PET, SPECT, spectral CT, phase contrast CT, electron tomography, . . .
- Mathematical structures/notions: Functional, operator, Fréchet derivative, proximal, diffeomorphism, discretization, sparsifying transforms, . . .
- Flexibility: Mathematical structures/notions re-usable across modalities
  $\rightsquigarrow$ Make it easy to "play around" with new ideas and combine concepts.
- 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, ...
- Mathematical structures/notions: Functional, operator, Fréchet derivative, proximal, diffeomorphism, discretization, sparsifying transforms, ...
- Flexibility: Mathematical structures/notions re-usable across modalities
  $\rightsquigarrow$ Make it easy to "play around" with new ideas and combine concepts.
- 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.

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.
- 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

An object-oriented Python framework for inverse problems

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.

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.

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.

Main components:

- Library of atomic mathematical components
  - Deformation operators
  - Function transforms: wavelet, Fourier, . . .
  - 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, . . .

Main components:

- Library of atomic mathematical components
  - Deformation operators
  - Function transforms: wavelet, Fourier, ...
  - 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, ...

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 3 major Deep Learning frameworks: **Tensorflow**, **Theano** and **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
- *Data space* Y
- *Forward operator* $\mathcal{T} : X \to Y$
- *Data* $g \in Y$
- *Data discrepancy functional* $\| \cdot - g\|_Y^2$
- *Regularization parameter* $\lambda > 0$
- *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 to mathematicians

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 to mathematicians

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 to mathematicians

Design principle: abstraction

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

$$f_{k+1} = f_k + \omega \mathcal{T}'(f_k)^* \big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

Code using ODL:

```
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:
  - `T(f)` $\longleftrightarrow$ $\mathcal{T}(f)$ (operator evaluation)
  - `T.derivative(f)` $\longleftrightarrow$ $\mathcal{T}'(f)$ (derivative *operator* at f)
  - `T.derivative(f).adjoint` $\longleftrightarrow$ $\mathcal{T}'(f)^*$ (adjoint of the derivative at f)

Design principle: abstraction

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

$$f_{k+1} = f_k + \omega \mathcal{T}'(f_k)^* \big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

Code using ODL:

```
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:
  - `T(f)` $\longleftrightarrow$ $\mathcal{T}(f)$ (operator evaluation)
  - `T.derivative(f)` $\longleftrightarrow$ $\mathcal{T}'(f)$ (derivative *operator* at f)
  - `T.derivative(f).adjoint` $\longleftrightarrow$ $\mathcal{T}'(f)^*$ (adjoint of the derivative at f)

Design principle: abstraction

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

$$f_{k+1} = f_k + \omega \mathcal{T}'(f_k)^* \big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

Code using ODL:

```
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:
  - `T(f)` $\longleftrightarrow$ $\mathcal{T}(f)$ (operator evaluation)
  - `T.derivative(f)` $\longleftrightarrow$ $\mathcal{T}'(f)$ (derivative *operator* at f)
  - `T.derivative(f).adjoint` $\longleftrightarrow$ $\mathcal{T}'(f)^*$ (adjoint of the derivative at f)

Design principle: abstraction

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

$$f_{k+1} = f_k + \omega \mathcal{T}'(f_k)^* \big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \dots, K-1$$

Code using ODL:

```
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:
  - `T(f)` $\longleftrightarrow \mathcal{T}(f)$ (operator evaluation)
  - `T.derivative(f)` $\longleftrightarrow \mathcal{T}'(f)$ (derivative *operator* at f)
  - `T.derivative(f).adjoint` $\longleftrightarrow \mathcal{T}'(f)^*$ (adjoint of the derivative at f)

Design principle: abstraction

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

$$f_{k+1} = f_k + \omega \mathcal{T}'(f_k)^* \big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

Code using ODL:

```
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: abstraction

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

$$f_{k+1} = f_k + \omega \mathcal{T}'(f_k)^* \big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \dots, K-1$$

Code using ODL:

```
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: abstraction

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

$$f_{k+1} = f_k + \omega \mathcal{T}'(f_k)^* \big(g - \mathcal{T}(f_k)\big), \quad k = 0, 1, \ldots, K-1$$

Code using ODL:

```
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`
- 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

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

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

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

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

Design compromises

- Distinguish types only if they have *computable* differences.

  Example: There is only one `FunctionSpace` representing $\mathcal{F}(\Omega, \mathbb{F}) = \{f : \Omega \to \mathbb{F}\}$.
  Reason: No way to do "interesting" things without symbolic calculus.

- Include only features that *add value* in a specific task.

  Example: Currently, only nearest neighbor and linear interpolation implemented.
  Reason: Higher-order schemes are seldom used in imaging, no need so far.

- Enforce mathematical rigor only to help the users. *Don't stand in their way.*

  Example: For operator composition `T * S`, check if `S.range == T.domain`.
  Conversely, no convexity check of `Functionals` in optimization methods.
  Reason: Wrong domains/ranges are a major error source, checking helps users.
  Conversely, a convex method may work well even on a non-convex problem.

Design compromises

- Distinguish types only if they have *computable* differences.

  Example: There is only one `FunctionSpace` representing $\mathcal{F}(\Omega, \mathbb{F}) = \{f : \Omega \to \mathbb{F}\}$.
  Reason: No way to do "interesting" things without symbolic calculus.

- Include only features that *add value* in a specific task.

  Example: Currently, only nearest neighbor and linear interpolation implemented.
  Reason: Higher-order schemes are seldom used in imaging, no need so far.

- Enforce mathematical rigor only to help the users. *Don't stand in their way.*

  Example: For operator composition `T * S`, check if `S.range == T.domain`.
  Conversely, no convexity check of `Functionals` in optimization methods.
  Reason: Wrong domains/ranges are a major error source, checking helps users.
  Conversely, a convex method may work well even on a non-convex problem.

Design compromises

- Distinguish types only if they have *computable* differences.

  Example: There is only one `FunctionSpace` representing $\mathcal{F}(\Omega, \mathbb{F}) = \{f : \Omega \to \mathbb{F}\}$.
  Reason: No way to do "interesting" things without symbolic calculus.

- Include only features that *add value* in a specific task.

  Example: Currently, only nearest neighbor and linear interpolation implemented.
  Reason: Higher-order schemes are seldom used in imaging, no need so far.

- Enforce mathematical rigor only to help the users. *Don't stand in their way*.

  Example: For operator composition `T * S`, check if `S.range == T.domain`.
  Conversely, no convexity check of `Functionals` in optimization methods.
  Reason: Wrong domains/ranges are a major error source, checking helps users.
  Conversely, a convex method may work well even on a non-convex problem.

Further design considerations

- ODL is a *prototyping* framework, not a black-box solution
  - ⇝ Give users freedom to experiment and tinker, do "unorthodox" things
  - ⇝ Very little "intelligence" that guesses what a user wants
  - ⇝ 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!
  - ⇝ Use libraries in the most efficient way possible (avoid copies, operate in-place, work with low-dimensional arrays, vectorization, broadcasting, . . .)
  - ⇝ Compute on the GPU whenever possible (new fast back-end coming soon)

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

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

Further design considerations

- ODL is a *prototyping* framework, not a black-box solution
  - ⇝ Give users freedom to experiment and tinker, do “unorthodox” things
  - ⇝ Very little “intelligence” that guesses what a user wants
  - ⇝ 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!
  - ⇝ Use libraries in the most efficient way possible (avoid copies, operate in-place, work with low-dimensional arrays, vectorization, broadcasting, . . .)
  - ⇝ 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 $\mathcal{R}\colon L^2(\Omega) \to Y$ defined as

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

for all lines $\ell = \{x = t\theta + v \in \mathbb{R}^2 \mid t \in \mathbb{R},\ \theta \in S^1,\ v \perp \theta\}$.

Given: Noisy data

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

Regularization: Conjugate gradient (CGLS) with early termination

Example: tomography

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

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

for all lines $\ell = \{x = t\theta + v \in \mathbb{R}^2 \mid t \in \mathbb{R},\ \theta \in S^1,\ v \perp \theta\}$.

Given: Noisy data

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

Regularization: Conjugate gradient (CGLS) with early termination

Example: tomography

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

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

for all lines $\ell = \{x = t\theta + v \in \mathbb{R}^2 \mid t \in \mathbb{R},\ \theta \in S^1,\ v \perp \theta\}$.

Given: Noisy data

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

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
- Create parallel beam geometry with P angles and K detector pixels
- Define ray transform $\mathcal{R}\colon L^2(\Omega) \to Y$ (the space Y is inferred from the geometry)
- Solve inverse problem using CGLS
- Display the results

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 $\mathcal{R} \colon L^2(\Omega) \to Y$ (the space Y is inferred from the geometry)
- Solve inverse problem using CGLS
- Display the results

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 $\mathcal{R}\colon L^2(\Omega) \to Y$ (the space Y is inferred from the geometry)
- Solve inverse problem using CGLS
- Display the results

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 $\mathcal{R} \colon L^2(\Omega) \to Y$ (the space Y is inferred from the geometry)
- Solve inverse problem using CGLS
- Display the results

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 $\mathcal{R}\colon L^2(\Omega) \to Y$ (the space Y is inferred from the geometry)
- Solve inverse problem using CGLS
- Display the results

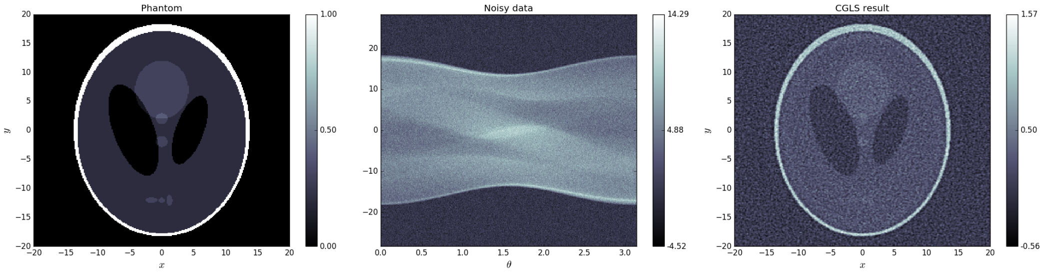
Example: Tomography

Code

```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, 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

Results



Example: Variable L^p TV denoising

Inverse Problem: Find $f_\alpha \in L^2(\Omega)$ such that

$$f_\alpha = \operatorname*{arg\,min}_{f \in L^2(\Omega)} \left[\|f - g\|_{L^2}^2 + \rho_p(\nabla f) \right]$$

for a given noisy image g. Here,

$$\rho_p(f) := \int_\Omega |f(x)|^{p(x)} \,\mathrm{d}x$$

for given $p : \Omega \to [1, 2]$.

Strategy: Use prox_{ρ_p} and a splitting method to find a minimizer [Koh17].

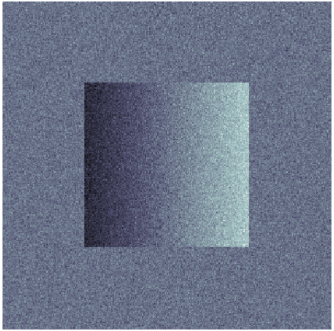
H. K.
Total variation regularization with variable Lebesgue prior.
arXiv:1702.08807 [math], February 2017.

Example: Variable L^p TV denoising

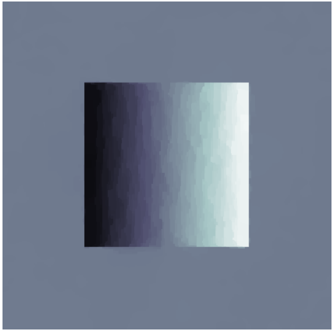
Code

```
# Read or generate data, Compute the exponent from the data
...

# Setup functionals and operators
data_matching = odl.solvers.L2NormSquared(reco_space).translated(data)
varlp_func = variable_lp.VariableLpModular(gradient.range, exponent)
regularizer = 2e-1 * varlp_func
constraint = odl.solvers.IndicatorBox(reco_space, -5, 5)

lin_ops = [odl.IdentityOperator(reco_space), gradient]

# Start iteration from the noisy data
x = data.copy()

# Choose optimization parameters and go
odl.solvers.douglas_rachford_pd(x, constraint, [data_matching, regularizer],
                                lin_ops, tau=tau, sigma=sigma, lam=lam,
                                niter=100)
```

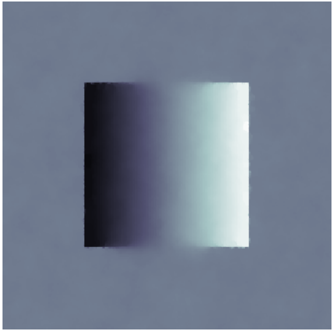
Example: Variable L^p TV denoising
Results



Noisy image



TV denoised



Variable L^p TV denoised

Work in progress

- Full multi-indexing support also on the GPU (using Theano's `libgpuarray`)

```
>>> space_cpu = odl.rn((2, 3, 4))  # NumPy backend
>>> x_cpu = space_cpu.zero()
>>> x_cpu[:, :3, ::2] = -5
>>> x_cpu[0]
rn((3, 4)).element(
    [[-5.,  0., -5.,  0.],
     [-5.,  0., -5.,  0.],
     [-5.,  0., -5.,  0.]]
)

>>> space_gpu = odl.rn((2, 3, 4), impl='gpuarray')  # libgpuarray backend
>>> x_gpu = space_gpu.zero()
>>> x_gpu[:, :3, ::2] = -5
>>> x_gpu[0]
rn((3, 4), impl='gpuarray').element(
    [[-5.,  0., -5.,  0.],
     [-5.,  0., -5.,  0.],
     [-5.,  0., -5.,  0.]]
)
```

Work in progress

- Vector- and tensor-valued functions, useful for, e.g., multi-channel problems or deformation fields

```
>>> # Vector-valued functions, 2 components
>>> space = odl.uniform_discr(0, 1, 5, dtype=(float, (2,)))
>>> def f(x):
...     return (x + 1, x ** 2 - 1)
>>> f_elem = space.element(f)
>>> f_elem  # Discretized using vectorized evaluation
uniform_discr(0.0, 1.0, 5, dtype=('float64', (2,))).element(
    [[ 1.1    1.3    1.5    1.7    1.9  ]
     [-0.99  -0.91  -0.75  -0.51  -0.19]]
)
```

Work in progress

- Deep integration with Deep Learning frameworks

```
>>> odl_op = odl.RayTransform(...)
>>> # Make a layer for Tensorflow
>>> tf_layer = odl.contrib.tensorflow.as_tensorflow_layer(odl_op)
>>> # Make a Theano operator
>>> theano_op = odl.contrib.theano.TheanoOperator(odl_op)
>>> # Make Torch autograd Function
>>> torch_op = odl.contrib.pytorch.TorchOperator(odl_op)
```

  Fully support backpropagation (automatic differentiation) via `operator.derivative(x).adjoint`