ODL, A python library for Inverse Problems

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

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

[2]Elekta

Operator Discretization Library

A easy to use python library which lets researchers in Inverse Problems write prototypes while exploiting optimized hardware and libraries in an uniform way and with little overhead.

Operator Discretization Library

A easy to use python library which lets researchers in Inverse Problems write prototypes while exploiting optimized hardware and libraries in an uniform way and with little overhead.

Operator Discretization Library

A easy to use python library which lets researchers in Inverse Problems write prototypes while exploiting optimized hardware and libraries in an uniform way and with little overhead.

Operator Discretization Library

A easy to use python library which lets researchers in Inverse Problems write prototypes while exploiting optimized hardware and libraries in an uniform way and with little overhead.

Operator Discretization Library

A easy to use python library which lets researchers in Inverse Problems write prototypes while exploiting optimized hardware and libraries in an uniform way and with little overhead.

Operator Discretization Library

A easy to use python library which lets researchers in Inverse Problems write prototypes while exploiting optimized hardware and libraries in an uniform way and with little overhead.

Operator Discretization Library

A easy to use python library which lets researchers in Inverse Problems write prototypes while exploiting optimized hardware and libraries in an uniform way and with little overhead.

Motivation

- Avoid duplicate work
- Glue different ideas together.
- Allow expression of math formulas and nomenclature relatively close to code
- Not everyone knows/wants to use C++/CUDA etc
- Reduce errors by reusing tested components

Intended Scope

- Tomography CT, CBCT, SPECT, PET, ... more?
- Mixed experience and preferences.
  - Easy to understand
  - Design should not be limiting
- Needs to be usable on clinical datasets
  - No unavoidable "$\mathcal{O}(n)$" overhead
  - Access to hardware acceleration (*GPU, MPI, Xeon Phi, ...*)
- Intended for prototyping, not clinical use
  - Minimal developer overhead
  - Easy interface with input (DICOM, ...) and output, plotting.

Components

- Core
  - Analysis, operators and spaces
  - Discretizations
    - Voxel, Fourier, Wavelet
    - CUDA, Xeon Phi, MPI
    - float, complex
- Tomography
  - Geometries, Cone beam, Parallel beam, SPECT
  - Solvers
  - Interfaces, STIR, ASTRA, NiftyRec, etc
  - Data input

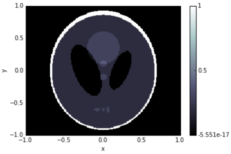
Example - Problem

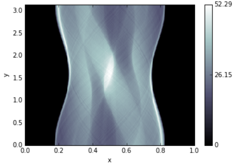
A: Radon transform

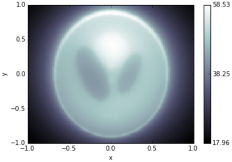
$$A : \mathcal{L}^2([-1,1] \times [-1,1]) \to \mathcal{L}^2([0,1] \times [0,\pi])$$

A^*: Back-projection

$$A^* : \mathcal{L}^2([0,1] \times [0,\pi]) \to \mathcal{L}^2([-1,1] \times [-1,1])$$







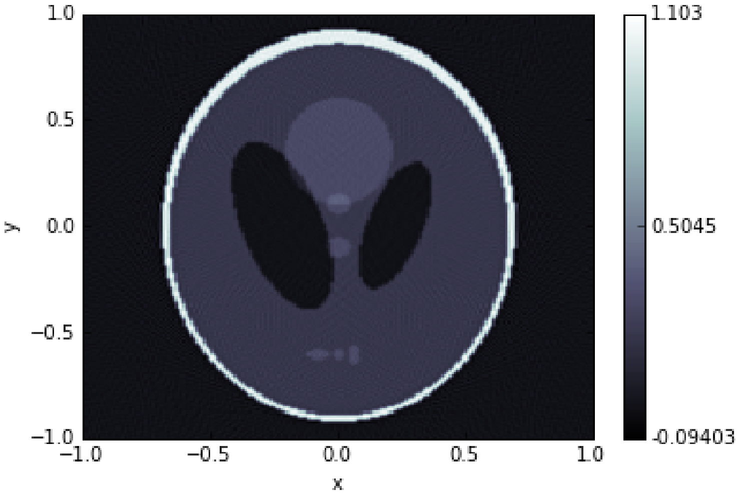
Example - Algorithm

Inverse problem $Ax = g$ solved with SART/landweber algorithm:

input : Operator A, data g
output: Reconstruction x_n
$\omega = 1/||A^* A||$
for $i = 1, \ldots, n$ **do**
  $x_i = x_{i-1} - \omega A^*(Ax_{i-1} - g)$
end

Example - Code

```
square = odl.L2(odl.Rectangle([-1, -1], [1, 1]))
sino = odl.L2(odl.Rectangle([0, 0], [1, pi]))

dom = odl.uniform_discr(square, [100, 100])
ran = odl.uniform_discr(sino, [142, 100])

A = ParallelForwardProjector(dom, ran)
omega = 1/odl.diagnostics.OperatorTest(A.adjoint * A).norm()

phantom = odl.util.shepp_logan(dom)
data = A(phantom)

x = domain.zero()
for i in range(iterations):
    x = x - omega * A.adjoint(A(x) - data)

x.show()
```


1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x
1.103
0.5045
-0.09403

Zero overhead SART

Simple

```
...
for i in range(iterations):
    x = x - omega * A.adjoint(A(x) - data)
```

Zero overhead SART

Simple

```
...
for i in range(iterations):
    x = x - omega * A.adjoint(A(x) - data)
```

0 - overhead

```
...
tmp1 = ran.element()
tmp2 = dom.element()
for i in range(iterations):
    A(recon, out=tmp1)
    tmp1 -= data
    A.adjoint(tmp1, out=tmp2)
    tmp2 *= omega
    x -= tmp2
```

Simplify writing algorithms

$$\min f(x) \implies \nabla f(x) = 0$$

Simplify writing algorithms

$$\min f(x) \Longrightarrow \nabla f(x) = 0$$

$H_0^{-1} = I$

for *i = 1, ..., n* **do**

$p_i = -H_{i-1}^{-1} \nabla f(x_{i-1})$

$\alpha_i = LineSearch(...)$

$s_i = \alpha_i p_i$

$x_i = x_{i-1} + s_i$

$y_i = \nabla f(x_i) - \nabla f(x_{i-1})$

$$H_i^{-1} = \left(I - \frac{sy^T}{y^T s}\right) H_{i-1}^{-1} \left(I - \frac{ys^T}{y^T s}\right) + \frac{ss^T}{y^T s}$$

end

Simplify writing algorithms

$$\min f(x) \Longrightarrow \nabla f(x) = 0$$

$H_0^{-1} = I$
for $i = 1, \ldots, n$ **do**

$$p_i = -H_{i-1}^{-1} \nabla f(x_{i-1})$$
$$\alpha_i = LineSearch(\ldots)$$
$$s_i = \alpha_i p_i$$
$$x_i = x_{i-1} + s_i$$
$$y_i = \nabla f(x_i) - \nabla f(x_{i-1})$$
$$H_i^{-1} = \left(I - \frac{sy^T}{y^T s}\right) H_{i-1}^{-1} \left(I - \frac{ys^T}{y^T s}\right) + \frac{ss^T}{y^T s}$$

end

```
Hi = I = IdentityOperator(gradf.range)
df = gradf(x)
for i in range(n):
  p = -Hi(df)
  alpha = line_search(x, p, df)
  s = alpha * p
  x = x + s

  df, df_old = gradf(x), df
  y = df - df_old

  ys = y.T(s)
  Hi = (I - s * y.T / ys) * Hi *
       (I - y * s.T / ys) + s * s.T / ys
```

Automatic tests

- Test projectors
  - Linearity
  - Matched forward/back projector
  - Scaling/norm
  - Derivative
- Test discretization
  - Norm axioms
  - Convergence

Performance

Comparsion with CGLS-CUDA[1]. Solve $Ax = b$ using CGN with CUDA[2].

Library	Runtime	Lines of code	Language
ODL	195s	20	CUDA/C++/Python
CGLS-CUDA	185s	700	CUDA

[1] https://github.com/foges/cgls_cuda
[2] $A \in \mathbb{R}^{10000 \times 10000}$, 10000 iterations on GTX980

Questions

Questions?

https://github.com/odlgroup/odl