CCPi Core Imaging Library (CIL): modular optimisation framework

Jakob Sauer Jørgensen

School of Mathematics & Henry Moseley X-ray Imaging Facility, U. Manchester

jakob.jorgensen@manchester.ac.uk

CCPPETMR software TCON

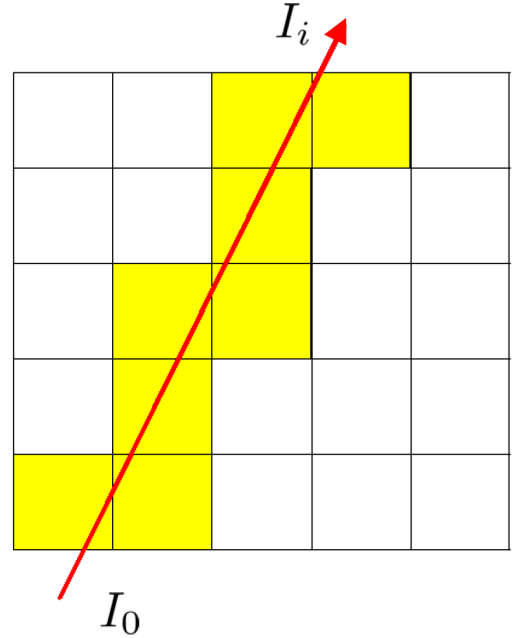
23 March 2018

Joint work with: **Daniil Kazantsev** (U. Manchester), **Edoardo Pasca** (STFC) and **Srikanth Nagella** (STFC)

Imaging model for iterative X-ray CT reconstruction

Beer-Lambert for ray i (line L_i):

$$\int_{L_i} \mu(s)ds = -\log \frac{I_i}{I_0} = b_i$$



Assume object constant in pixels:

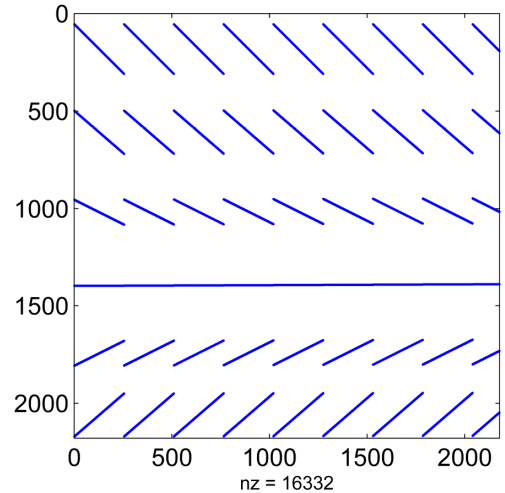
- x_j is the jth pixel value.
- a_{ij} is path length through jth pixel.

Approximate line integral by sum:

$$\sum_j a_{ij} x_j = b_i$$

Large sparse linear system:

$$Ax = b$$



Structure of system matrix A

Optimisation-based reconstruction

Common form traditionally:

- Smooth (differentiable) optimisation
- Data fidelity plus regularisation (prior)
- Example: Tikhonov

$$x^\star = \operatorname*{argmin}_x \frac{1}{2}\|Ax - b\|_2^2 + \alpha^2 \|Kx\|_2^2$$

- Algorithms evaluate function value, gradient, maybe Hessian.

Recent focus on non-smooth problems:

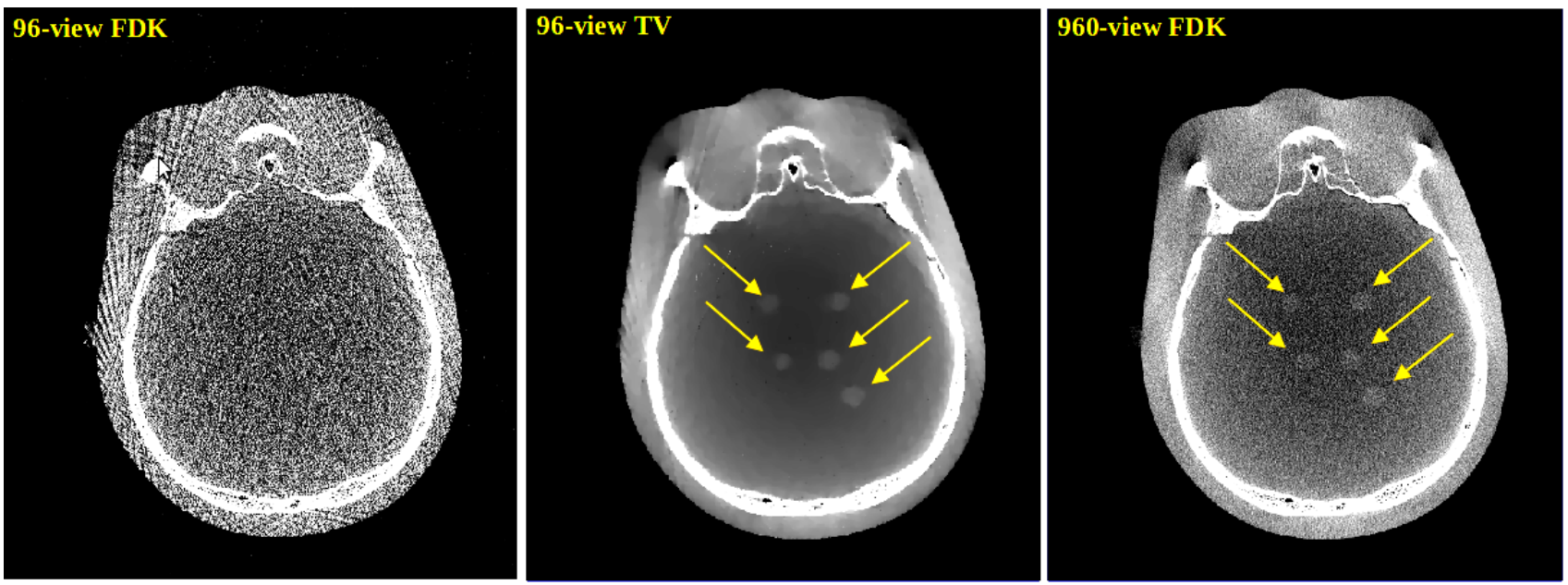
- Examples: Sparse regularisation such as 1-norm and TV type.
- Other forms: Multiple terms as well as constraints possible.
- Other algorithms needed.

Example: Total variation (TV) regularisation

Total variation: Homogeneous regions with sharp boundaries.

$$x^\star = \operatorname*{argmin}_x \|x\|_{\mathsf{TV}} \quad \text{such that} \quad \|Ax - b\|_2 \leq \epsilon$$

$$\|x\|_{\mathsf{TV}} = \sum_j \|D_j x\|_2, \quad D_j \text{ finite diff. gradient at voxel } j.$$



[Bian et al. 2010, Phys. Med. Biol. **55**, 6575–6599]. Courtesy: X. Pan, U. Chicago.

CCPi CIL Modular optimisation framework

Core classes similar to SIRF:

- `AcquisitionData` holding measured data.
- `ImageData` holding reconstructed images (and simulated test images).
- `Operator`: Representing (so far linear) operators, including the imaging model, i.e. SIRF AcquisitionModel.
- Supports multichannel datasets, with channels representing energies (spectral CT), times (temporal/dynamic CT), etc.
- Supports object arithmetics, e.g., addition of `ImageData` etc.

Optimisation framework:

- Also object oriented.
- Inspired by `http://proximity-operator.net/`

Operator class

- Typically represents the forward model (Acquisition model)
- Can also represent other linear operators like finite differencing useful for different kinds of regularization incl TV.
- Method `direct`: Evaluates direct (forward) application.
- Method `adjoint`: Evaluates adjoint (backward) application.
- Method `get_max_sing_val`: Computes and hold in object instance the largest singular value often needed in algorithms.
- Attribute `size`: Holds the size (dimensions) of operator.
- Can wrap third party software (currently ASTRA and own CCPi projectors) and provide uniform interface.

Generic problem

Consider generic two-term problem:

$$x^\star = \operatorname*{argmin}_x f(x) + g(x)$$

- $f(x)$ differentiable
- $g(x)$ non-differentiable, but with "easy" proximal operator

Examples:

- $f(x) = \frac{1}{2}\|Ax - b\|_2^2$
- $g(x) = \alpha\|x\|_1$
- $g(x) = \alpha\|x\|_{\mathrm{TV}}, \qquad \|x\|_{\mathrm{TV}} = \sum_j \|D_j\|_2$, where D_j is a finite difference operator at pixel j.

Function objects

Define methods/attributes reflecting properties of function:

- Differentiable: define function value and gradient
- Non-differentiable: no gradient, but proximal operator

Differentiable example:

- $f(x) = c\|Ax - b\|_2^2$
- Given operator `A`, data `b`, create: `f = Norm2sq(A,b,c=0.5)`
- Eval function at `xk`: `f.fun(xk)`
- Eval gradient at `xk`: `f.grad(xk)`

Non-differentiable example:

- $f(x) = \alpha\|x\|_1$
- Create: `g = Norm1(alph)`
- Eval function at `xk`: `g.fun(xk)`
- Eval prox at `xk`: `f.prox(xk)`

The function class, least squares

```
class Norm2sq(BaseFunction):

    def __init__(self,A,b,c=1.0):
        self.A = A
        self.b = b
        self.c = c
        self.L = 2.0 * self.c * \
                 (self.A.get_max_sing_val()**2)
        super(Norm2sq,self).__init__()

    def grad(self,x):
        return 2.0 * self.c * \
               self.A.adjoint(self.A.direct(x)-self.b)

    def fun(self,x):
        return self.c * \
               (((self.A.direct(x)-self.b)**2).sum())
```

The function class, the 1-norm

```
class Norm1(BaseFunction):

    def __init__(self,gamma):
        self.gamma = gamma
        self.L = 1
        super(Norm1,self).__init__()

    def fun(self,x):
        return self.gamma * (x.abs().sum())

    def prox(self,x,tau):
        return (x.abs() - tau*self.gamma).maximum(0) * \
               x.sign()
```

Generic algorithm: FISTA (Beck & Teboulle, 2009)

Given initial guess `x_init` and functions `f` and `g`:

```
x_old = x_init
y = x_init;
invL = 1/f.L
t_old = 1
for it in range(0, max_iter):
    u = y - invL*f.grad(y)
    x = g.prox(u,invL)
    t = 0.5*(1 + numpy.sqrt(1 + 4*(t_old**2)))
    y = x + (t_old-1)/t*(x-x_old)
    x_old = x
    t_old = t
```

Result:

`x` converges to the solution $x^\star$, i.e., the minimizer of $f(x) + g(x)$.

Another generic problem and an algorithm

Consider generic three-term problem:

$$x^\star = \operatorname*{argmin}_x f(x) + g(x) + h(Kx)$$

Function properties:

- g differentiable
- $h(Kx)$: h nondifferentiable, K operator
- f: nondifferentiable

Algorithm FBPD:

- Solvable by the Forward Backward Primal Dual algorithm.
- Useful when prox op. for $h(Kx)$ is hard, but for $h(\cdot)$ is easy.
- Extra term can for example encode constraints.

(Condat, 2013)

Live demo

Fingers crossed it will work...

Advantages of modular optimisation framework

- Separate definitions of different parts of optimisation problem, optimisation algorithm, operators, ...
- Code easier to write, and to maintain and test.
- Easy "prototyping": experiment with different reconstruction formulations and algorithms, i.e., "mix and match" data fidelities, regularizers and constraints.
- Can use different algorithms on same reconstruction problem, to verify results and compare speed.
- Easy to add new data fidelities, regularizers, algorithms.

Questions

- Do we need an additional "Objective function" class?
- How should basic/expert user use the software? Maybe allow experts to specify functions f, g etc. and algorithm, but provide "packages" for use without knowing the details.
- How efficient is this? Some overhead is the price of the convenience...
- Currently simply holding numpy arrays, thinking about other internal representation like C++ or GPU arrays with only pointers passed around.