(And backward!)

Efficient Forward Models for Image Reconstruction

Matthew Muckley

Meta AI

Declaration of
Financial Interests or Relationships

Speaker Name: Matthew Muckley

I have the following financial interest or relationship to disclose with regard to the subject matter of this presentation:

Company Name: Meta
Type of Relationship: Employee

The Setting

✅ You understand the basics of MRI physics
✅ Programmed a sequence and collected interesting data
✅ Have a mathematical model and procedure for reconstruct the data

? How do you implement your procedure on a computer so it reconstructs in time to finish your Ph.D.?

The task

- Reconstruct Non-Cartesian radial liver data with compressed sensing

Magnetic Resonance in Medicine 72:707–717 (2014)

Golden-Angle Radial Sparse Parallel MRI: Combination of Compressed Sensing, Parallel Imaging, and Golden-Angle Radial Sampling for Fast and Flexible Dynamic Volumetric MRI

Li Feng,[1,2]* Robert Grimm,[3] Kai Tobias Block,[1] Hersh Chandarana,[1] Sungheon Kim,[1,2] Jian Xu,[4] Leon Axel,[1,2] Daniel K. Sodickson,[1,2] and Ricardo Otazo[1,2]

Materials for this presentation

Non-Cartesian GRASP Data

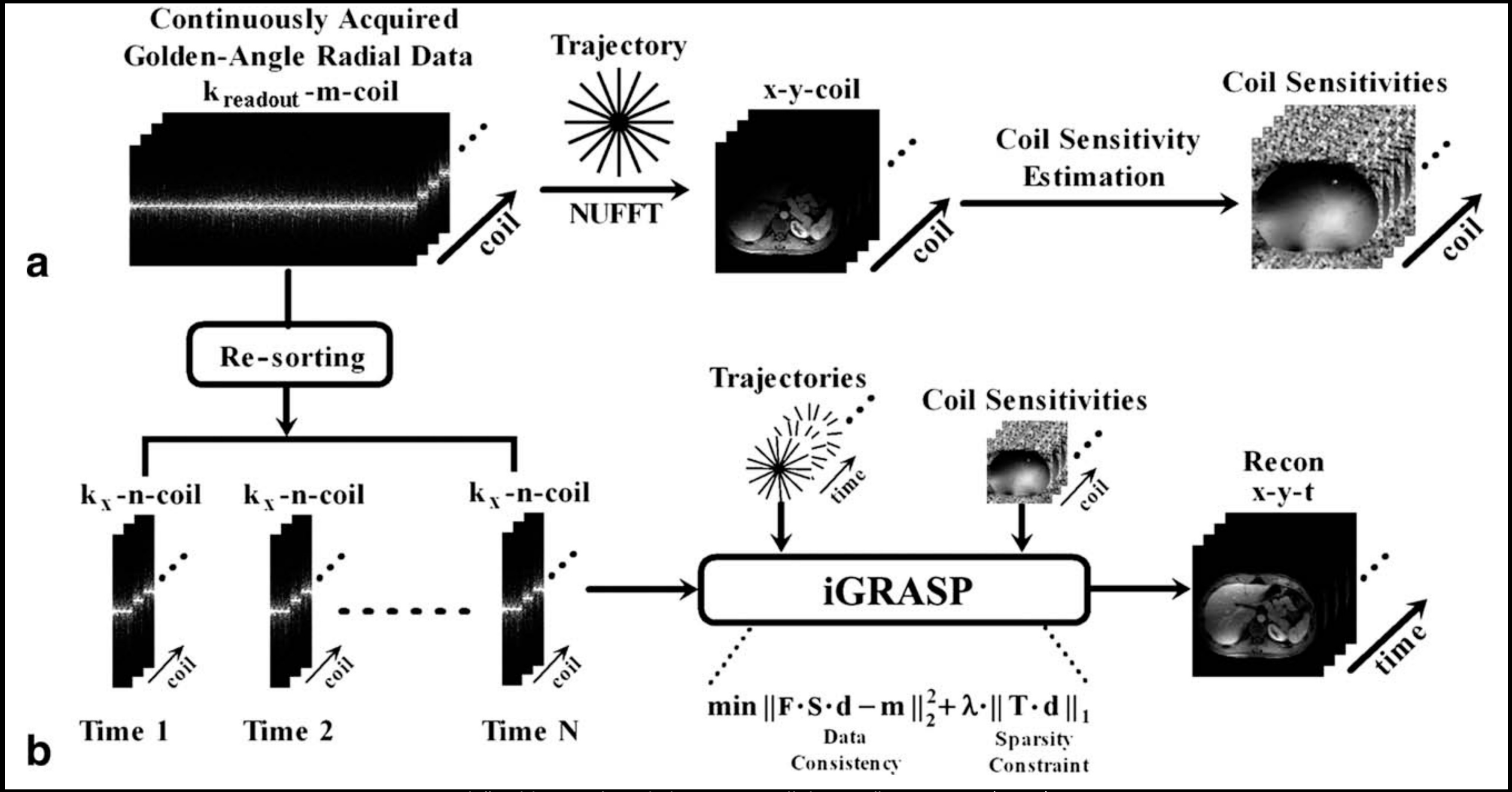
Feng, Li, et al. "Golden-angle radial sparse parallel MRI..." *MRM* 72.3 (2014): 707-717.

Code

If you find a mistake, raise an Issue!
If you have a question, start a Discussion!

Step 1: Understand Your Data

Golden Angle Radial Sparse Data



Feng, Li, et al. "Golden-angle radial sparse parallel MRI..." *MRM* 72.3 (2014): 707-717.

Inspecting Data Properties

```python
import yaml
import scipy.io as sio
import numpy as np

# load the data
with open("../data_loc.yaml", "r") as f:
    data_file = yaml.safe_load(f)

raw_data = sio.loadmat(data_file)

print(raw_data.keys())
print(raw_data["kdata"].shape)
print(raw_data["k"].shape)
ktraj = raw_data["k"]
print(f"real k.min(): {np.real(ktraj).min()}, real k.max(): {np.real(ktraj).max()}")
print(f"imag k.min(): {np.imag(ktraj).min()}, imag k.max(): {np.imag(ktraj).max()}")
```

✓ 0.6s

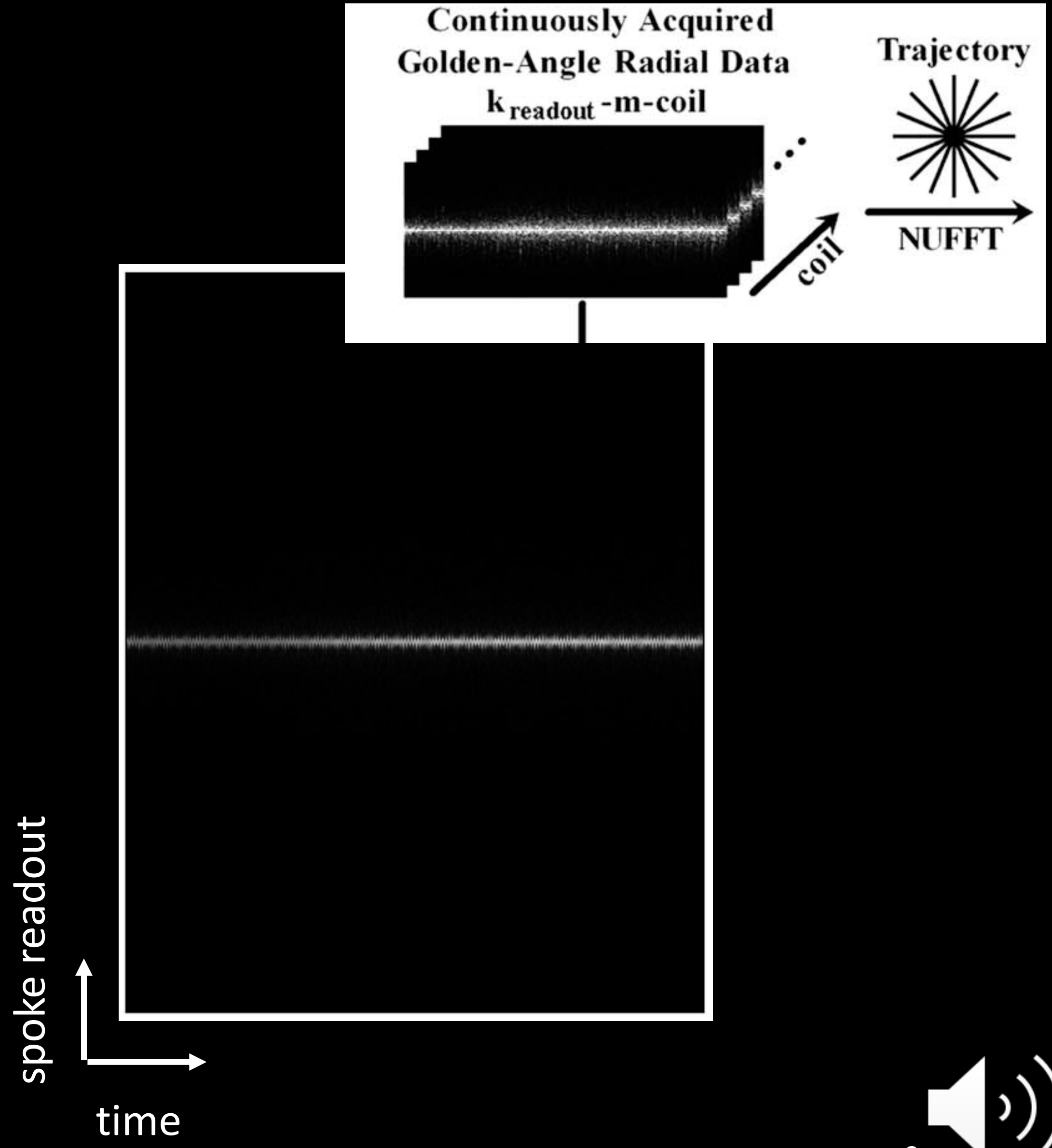
```
dict_keys(['__header__', '__version__', '__globals__', 'b1', 'kdata', 'k', 'w'])
(768, 600, 12)
(768, 600)
real k.min(): -0.49934862721385653, real k.max(): 0.49934862721385653
imag k.min(): -0.4993489583333333, imag k.max(): 0.4993489583333333
```

Plotting the Data

```
import matplotlib.pyplot as plt
import numpy as np

print(raw_data['kdata'].dtype)
plt.imshow(np.absolute(raw_data["kdata"][:, :, 0]))
plt.xticks([])
plt.yticks([])
plt.gray()
```

✓ 0.1s

```
complex128
```



Other things to check

Look through the header

- Number of coils
- Pulse sequence parameters (TE, TR)
- Slice thickness
- Readout filtering
- RF attributes (spoiling, etc.)

Step 2: Mapping the Reconstruction Algorithm

Objective

Solve the following (rewritten) compressed sensing optimization :

$$\hat{x} = \underset{x}{\operatorname{argmin}} \underbrace{||FSx - b||_2^2}_{\text{data consistency}} + \underbrace{\lambda ||Tx||_1}_{\text{sparsity regularization}}$$

Objective

Solve the following (rewritten) compressed sensing optimization :

(cost in the paper)

$$\hat{x} = \underset{x}{\operatorname{argmin}} \|FSx - b\|_2^2 + \lambda \|Tx\|_1$$

(cost in the code)

$$\hat{x} = \underset{x}{\operatorname{argmin}} \|FSx - b\|_W^2 + \lambda \|Tx\|_1$$

Picking a solver

$$\hat{x} = \underset{x}{\operatorname{argmin}} \|FSx - b\|_W^2 + \lambda \|Tx\|_1$$

Many options

- Non-linear conjugate gradient
  - Requires “corner-rounding”
- Augmented Lagrangian / ADMM / Split-Bregman
- Primal-dual

Picking a solver

$$\hat{x} = \underset{x}{\operatorname{argmin}} \|FSx - b\|_2^2 + \lambda \|Tx\|_1$$

Many options

- Non-linear conjugate gradient
  - Requires "corner-rounding"
- Augmented Lagrangian / ADMM / Split-Bregman
- **Primal-dual**

MFISTA

MFISTA

Input: $L \geq L(f)$—An upper bound on the Lipschitz constant of ∇f.

Step 0. Take $\mathbf{y}_1 = \mathbf{x}_0 \in \mathbb{E}, t_1 = 1$.

Step k. $(k \geq 1)$ Compute

$$\mathbf{z}_k = p_L(\mathbf{y}_k),$$

$$t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2} \tag{5.2}$$

$$\mathbf{x}_k = \operatorname{argmin}\{F(\mathbf{x}) : \mathbf{x} = \mathbf{z}_k, \mathbf{x}_{k-1}\} \tag{5.3}$$

$$\mathbf{y}_{k+1} = \mathbf{x}_k + \left(\frac{t_k}{t_{k+1}}\right)(\mathbf{z}_k - \mathbf{x}_k) + \left(\frac{t_k - 1}{t_{k+1}}\right)(\mathbf{x}_k - \mathbf{x}_{k-1}) \tag{5.4}$$

- $p_L(y) = \frac{1}{L} A'(Ay - b)$
- Minimizing $F(x)$ requires iterations with regularizer T

Beck, Amir, and Marc Teboulle. "Fast gradient-based…*IEEE-TIP* 18.11 (2009): 2419-2434.

MFISTA

MFISTA

Input: $L \geq L(f)$—An upper bound on the Lipschitz constant of ∇f.

Step 0. Take $\mathbf{y}_1 = \mathbf{x}_0 \in \mathbb{E}, t_1 = 1.$

Step k. $(k \geq 1)$ Compute

$$\mathbf{z}_k = p_L(\mathbf{y}_k),$$

$$t_{k+1} = \frac{1 + \sqrt{1 + 4t_k^2}}{2} \tag{5.2}$$

$$\mathbf{x}_k = \operatorname{argmin}\{F(\mathbf{x}) : \mathbf{x} = \mathbf{z}_k, \mathbf{x}_{k-1}\} \tag{5.3}$$

$$\mathbf{y}_{k+1} = \mathbf{x}_k + \left(\frac{t_k}{t_{k+1}}\right)(\mathbf{z}_k - \mathbf{x}_k) + \left(\frac{t_k - 1}{t_{k+1}}\right)(\mathbf{x}_k - \mathbf{x}_{k-1}) \tag{5.4}$$

Need forward/backward operators for physics

Need forward/backward operators for regularizer

Beck, Amir, and Marc Teboulle. "Fast gradient-based…*IEEE-TIP* 18.11 (2009): 2419-2434.

Implementation process

1. Formulate the forward operation based on physics
2. Derive adjoint (i.e., “backward”) operation based on forward operation
3. Decide on other regularizers and their corresponding operators for (1) and (2)
4. Estimate step sizes for all operators and implement optimization algorithm

Step 3: Implementing the data consistency model

The forward data model

$$b = FSx_{true} + e$$

- F: Non-Cartesian Fourier operator
- S: Sensitivity map operator
- x_{true}: Ground-truth image
- e: Gaussian noise

Sensitivity maps, $F\boxed{S}x$

Mathematics for C coils

$$Sx_{true} = \begin{bmatrix} s_1 \\ \cdots \\ s_C \end{bmatrix} x_{true}$$

each $s_1 \ldots, s_2$ is a diagonal, complex matrix with 1 sensitivity map

```python
print(f"image shape: {xtrue.shape}, sense maps shape: {sensitivity_maps.shape}")

output = []
for sensitivity_map in sensitivity_maps:
    output.append(xtrue * sensitivity_map)

output = np.stack(output)
print(f"output shape: {output.shape}")
```

✓ 0.0s

```
image shape: (384, 384), sense maps shape: (12, 384, 384)
output shape: (12, 384, 384)
```

Strategy 1 – for loop (slowest)

```
# operation 1, slowest
output = []
for sensitivity_map in sensitivity_maps:
    output.append(xtrue * sensitivity_map)

output1 = np.stack(output)
print(
    f"A shape: {sensitivity_maps.shape}, B shape: {xtrue.shape}, "
    f"output shape: {output1.shape}"
)
```

✓ 0.0s

```
A shape: (12, 384, 384), B shape: (384, 384), output shape: (12, 384, 384)
```

Loop and multiply each coil in loop

Strategy 2 – array copy (faster)

```
# operation 2, better
xtrue_expand = xtrue[None, ...]
xtrue_copy = np.repeat(xtrue_expand, 12, axis=0)
output2 = sensitivity_maps * xtrue_copy
print(
    f"A shape: {sensitivity_maps.shape}, B shape: {xtrue_copy.shape}, "
    f"output shape: {output2.shape}"
)
```

✓ 0.0s

```
A shape: (12, 384, 384), B shape: (12, 384, 384), output shape: (12, 384, 384)
```

Copy, then multiply in one operation

Strategy 3 – broadcasting (fastest, best)

```python
# operation 3, fasteset
output3 = sensitivity_maps * xtrue_expand
print(
    f"A shape: {sensitivity_maps.shape}, B shape: {xtrue_expand.shape}, "
    f"output shape: {output3.shape}"
)
```

Single, efficient multiply

✓ 0.0s

```
A shape: (12, 384, 384), B shape: (1, 384, 384), output shape: (12, 384, 384)
```

Speed comparison

```python
num_tests = 10000
op_speeds = {}
for ind, op in zip(range(1, 4), [op1, op2, op3]):
    start_time = time.perf_counter()
    for _ in range(num_tests):
        output = op(sensitivity_maps, xtrue)
    end_time = time.perf_counter()

    op_speed = (end_time - start_time) / num_tests
    print(f"op{ind} speed: {op_speed} seconds")
```

✓ 2m 44.6s

loop
copy
broadcast

```
op1 speed: 0.0064267619041005674 seconds
op2 speed: 0.005319759250000061 seconds
op3 speed: 0.004713306575000025 seconds
```

Broadcasting explanation

- Broadcasting can be applied to product operations
- Example: multiply each C axis of tensor A of size (C, H, W) with tensor B of size (H, W)

Broadcasting solution

- Extremely fast, minimizes memory copies
- Available in most computational languages (Numpy, MATLAB, Julia)

- Broadcasting very efficient for block-column matrices with diagonal blocks (e.g., SENSE)

Sensitivity maps, $\boxed{S'}F'b$

- Forward operation

$$Sx_{true} = \begin{bmatrix} s_1 \\ \cdots \\ s_C \end{bmatrix} x_{true}$$

```python
# build the adjoint operation
def op_adjoint(sensitivity_maps, fy):
    return np.sum(np.conj(sensitivity_maps) * fy, axis=0)
```

- Adjoint operation

$$S'F'y = [s_1', \ldots, s_C']x_{true}$$

Adjoint tests

- Adjoint property

$$\langle Sh_1, h_2 \rangle = \langle h_1, S'h_2 \rangle$$

- Testing with random numbers will catch (most) errors

```python
# build the adjoint operation
def op_adjoint(sensitivity_maps, fy):
    return np.sum(np.conj(sensitivity_maps) * fy, axis=0)

output = op_adjoint(sensitivity_maps, output)

# test the adjoint operation
im_shape = (1, xtrue.shape[-2], xtrue.shape[-1])
coil_im_shape = (sensitivity_maps.shape[0], xtrue.shape[-2], xtrue.shape[-1])
vec1 = np.random.normal(size=im_shape) + 1j*np.random.normal(size=im_shape)
vec2 = np.random.normal(size=coil_im_shape) + 1j*np.random.normal(size=coil_im_shape)

def complex_tensor_inprod(a, b):
    return np.sum(np.conj(a) * b)

inprod1 = complex_tensor_inprod(op3(sensitivity_maps, vec1), vec2)
inprod2 = complex_tensor_inprod(vec1, op_adjoint(sensitivity_maps, vec2))

print(np.allclose(inprod1, inprod2))
```

✓ 0.1s Py

Non-Cartesian Fourier operator, $\boxed{F}Sx$

Non-Cartesian Fourier operator, $\boxed{F}Sx$

- General Non-Cartesian Fourier operator for F (very slow):

$$b_c(k_m) = \sum_{n=0}^{N} \tilde{x}_{c,n}\, e^{-ik_m n}$$

- Interpolated (NUFFT) operation:

$$\boxed{q_{c,l} = \sum_{n=0}^{N} g_n\, \tilde{x}_{c,n} e^{-i\gamma l n}}$$

Oversampled FFT with scaling coefficients g_n

$$\boxed{b_c(k_m) = \sum_{j=1}^{J} q_{c,\{l_m+j\}_L} u_j(k_m)}$$

Interpolation to off-grid points with J-size interpolation kernel

Fessler, J.A., and Sutton, B.P. "Nonuniform fast Fourier transforms..." *IEEE transactions on signal processing* 51.2 (2003): 560-574.

NUFFT Implementations (a partial list)

- Min/max NUFFT (Fessler and Sutton, 2003)
  - Available in MIRT (MATLAB) and MIRT.jl (Julia)
  - PyNUFFT (Python)
- Kaiser-Bessel NUFFT
  - sigpy (Python)
  - torchkbnufft (Python, PyTorch)
  - TF KB-NUFFT (Python, Tensorflow)
  - gpuNUFFT (MATLAB, C++)
- Super-fast Gaussian NUFFT (Barnett et al., 2019)
  - Available in Flatiron NUFFT (FINUFFT) (C++, wrappers for Python)
  - Wrapper for FINUFFT in NFFT.jl (Julia)

NUFFT with PyTorch/torchkbnufft

- torchkbnufft: completely high-level
  - Easy to install, run on multiple devices
  - Slower speed than several other options
- Create high-level NUFFT object for Kaiser-Bessel interpolation

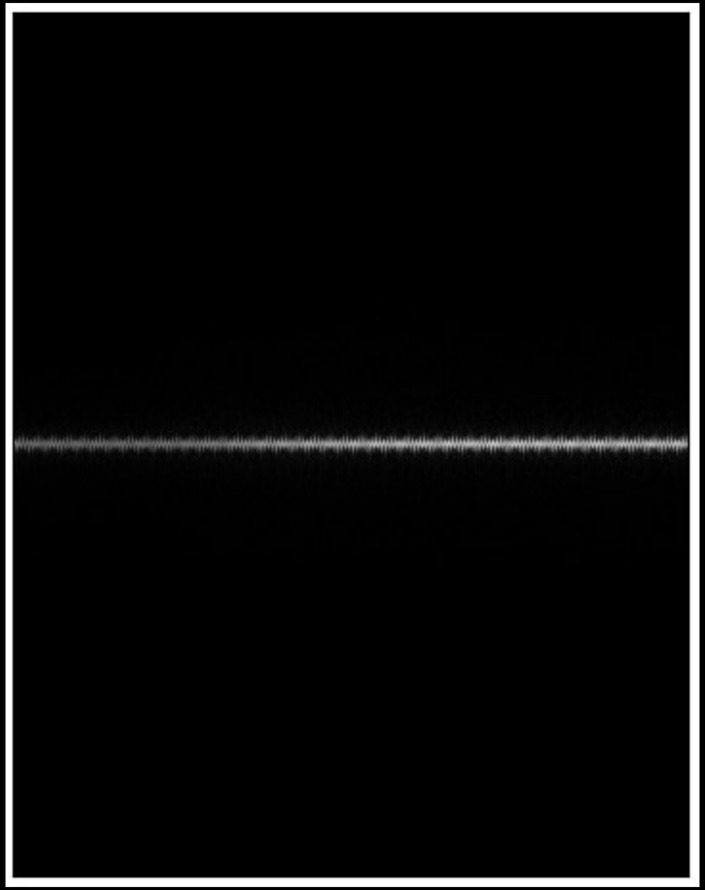
```python
import torch
from torchkbnufft import KbNufft

# extract the k-space trajectory
ktraj = raw_data["k"]

# build the NUFFT object
nufft_ob =KbNufft(im_size=(xtrue.shape[-2], xtrue.shape[-1]))

# torchkbnufft expects ktraj in radians/voxel
ktraj_torch = ktraj * 2 * np.pi
# stack the spatial dimensions instead of have real/imag
ktraj_torch = np.stack((np.real(ktraj_torch), np.imag(ktraj_torch)))

# convert to PyTorch tensor
ktraj_torch = torch.tensor(ktraj_torch, dtype=torch.float32).reshape(2, -1).contiguous()
# PyTorch expects batch dimension
coil_images_torch = torch.tensor(xtrue, dtype=torch.complex64).unsqueeze(0).unsqueeze(0)

with torch.no_grad():
    data = nufft_ob(coil_images_torch, ktraj_torch)
print(data.shape)
```

✓ 2.7s Python

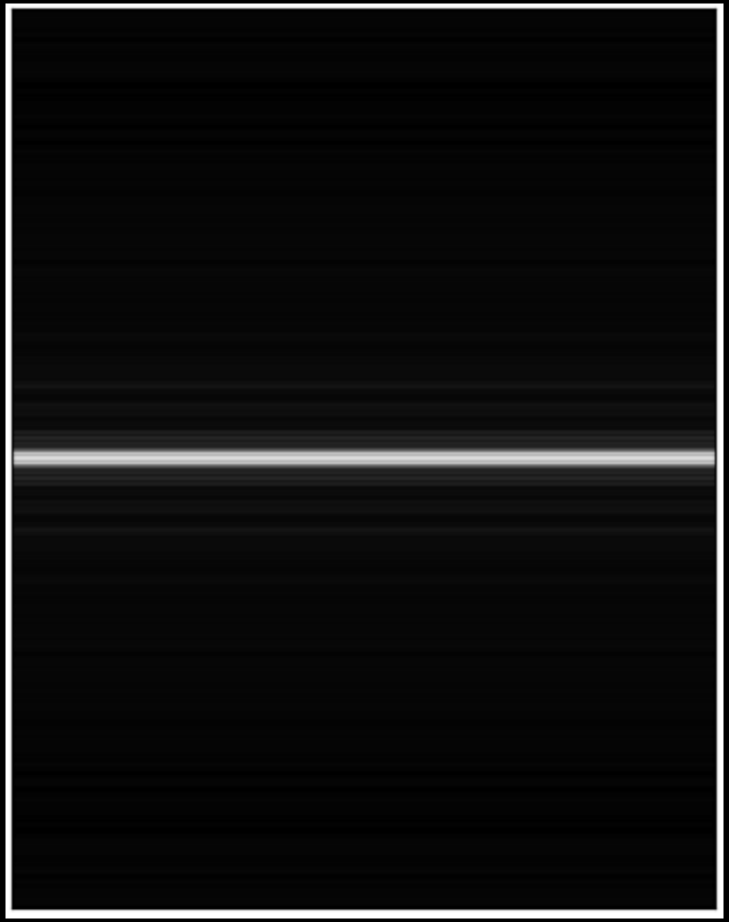
```
torch.Size([1, 1, 460800])
```

Verifying NUFFT output

Raw data



Shepp-Logan phantom



NUFFT adjoint in torchkbnufft

- Instantiate new adjoint NUFFT object
- Run adjoint object with same k-space trajectory

```
from torchkbnufft import KbNufftAdjoint

# instantiate adjoint object
adj_ob = KbNufftAdjoint(im_size=(xtrue.shape[-2], xtrue.shape[-1]))

# run the adjoint
recon_image = adj_ob(data, ktraj_torch)
print(recon_image.shape)
```

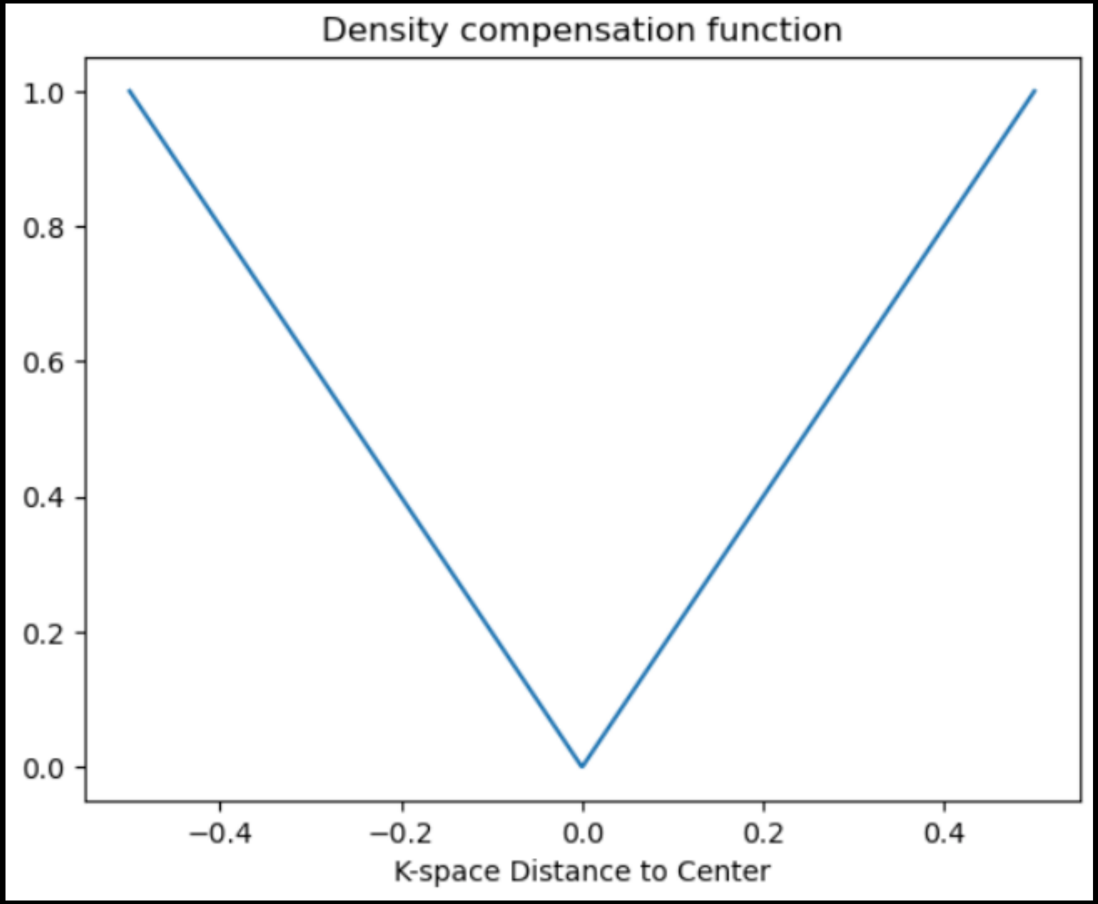
✓ 0.6s

```
torch.Size([1, 1, 384, 384])
```

Compare original image vs. reconstructed

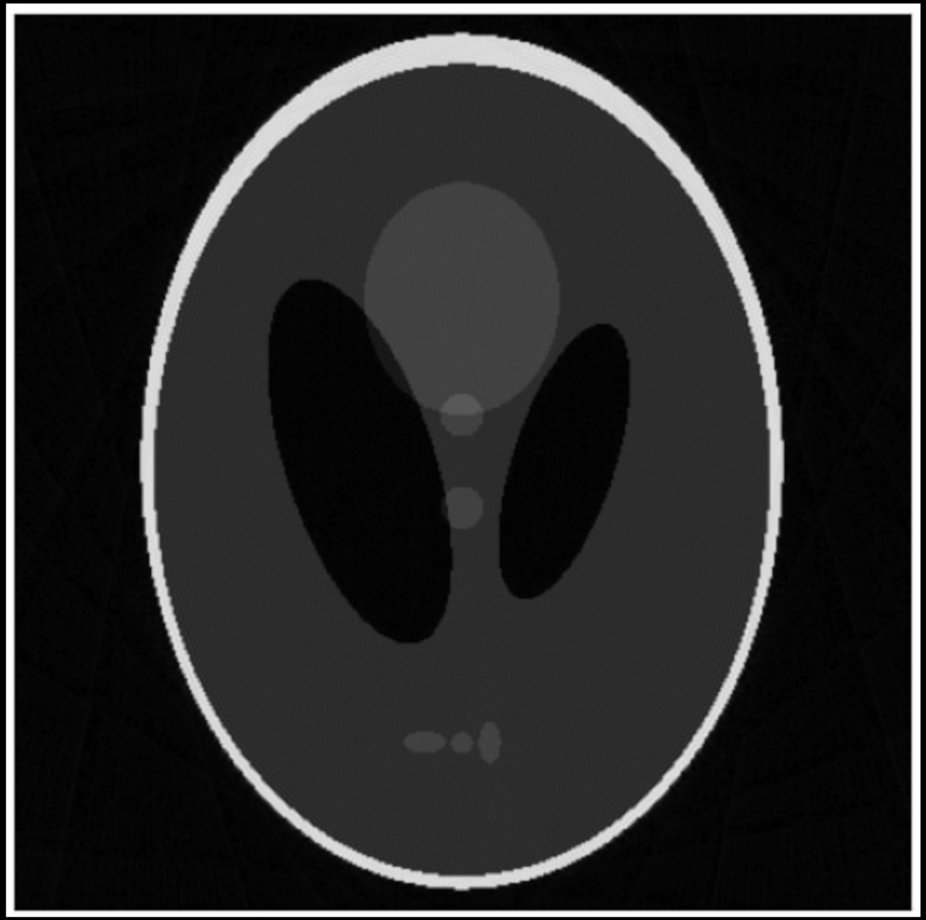
Density compensation

- With a radial trajectory, the center is sampled extra, but not compensated for
- Can use ramp weighting for radial



Density Compensation

```python
# pull out density compensation, unsqueeze batch/coil dimension
dcomp = torch.tensor(raw_data["w"], dtype=torch.float32).reshape(1, 1, -1)

# reconstruct with density compensation
recon_image = adj_ob(dcomp * data, ktraj_torch)
```



Step 4: Implementing the regularizer

Implementing the regularizer

- Finite differences forward

$$\begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & & \\ & & & -1 & 1 \end{bmatrix}$$

- Finite differences adjoint

$$\begin{bmatrix} -1 & & & \\ 1 & -1 & & \\ & & \ddots & \\ & & 1 & -1 \\ & & & 1 \end{bmatrix}$$

```python
def finite_forward(mat):
    return mat[1:] - mat[:-1]

def finite_adjoint(mat):
    return torch.cat(
        (
            mat[0].unsqueeze(0) * -1,
            mat[:-1] - mat[1:],
            mat[-1].unsqueeze(0)
        )
    )
```

Step 5: convenience wrapping and initial estimate

Linear operators for convenience

```python
import torch
from torch import Tensor

from ._linop import LinearOperator


class FiniteDifference(LinearOperator):
    def forward(self, image: Tensor) -> Tensor:
        # assume input is of size (num_timepoints, num_coils, ny, nx)
        return image[1:] - image[:-1]

    def adjoint(self, diffs: Tensor) -> Tensor:
        # assume input is of size (num_timepoints-1, num_coils, ny, nx)
        return torch.cat(
            (diffs[0].unsqueeze(0) * -1, diffs[:-1] - diffs[1:], diffs[-1].unsqueeze(0))
        )
```

Linear operators for convenience

```python
class LinearOperator(nn.Module):
    def forward(self, y):
        raise NotImplementedError

    def adjoint(self, y):
        raise NotImplementedError

```

```python
import torch
from torch import Tensor

from ._linop import LinearOperator


class FiniteDifference(LinearOperator):
    def forward(self, image: Tensor) -> Tensor:
        # assume input is of size (num_timepoints, num_coils, ny, nx)
        return image[1:] - image[:-1]

    def adjoint(self, diffs: Tensor) -> Tensor:
        # assume input is of size (num_timepoints-1, num_coils, ny, nx)
        return torch.cat(
            (diffs[0].unsqueeze(0) * -1, diffs[:-1] - diffs[1:], diffs[-1].unsqueeze(0))
        )
```

```python
from torch import Tensor
from torchkbnufft import KbNufft, KbNufftAdjoint

from ._linop import LinearOperator


class SenseNufftOp(LinearOperator):
    _sensitivity_maps: Tensor
    _ktraj: Tensor

    def __init__(self, sensitivity_maps: Tensor, ktraj: Tensor):
        super().__init__()
        self.register_buffer("_sensitivity_maps", sensitivity_maps)
        self.register_buffer("_ktraj", ktraj)

        im_size = (sensitivity_maps.shape[-2], sensitivity_maps.shape[-1])
        self._kbnufft = KbNufft(im_size=im_size)
        self._kabnufftadjoint = KbNufftAdjoint(im_size=im_size)

    def forward(self, image: Tensor) -> Tensor:
        # assume input is (num_timepoints, num_coils, ny, nx)
        return self._kbnufft(image, self._ktraj, smaps=self._sensitivity_maps)

    def adjoint(self, data: Tensor) -> Tensor:
        # assume input is (num_timepoints, num_coils, num_kspace)
        return self._kabnufftadjoint(data, self._ktraj, smaps=self._sensitivity_maps)
```

Running on the GPU with PyTorch

```
device = torch.device("cuda")
x = x.to(device)
x = x * 5
```

✓ 0.1s

```
data_op = data_op.to(device)
output = data_op.forward(x)
```

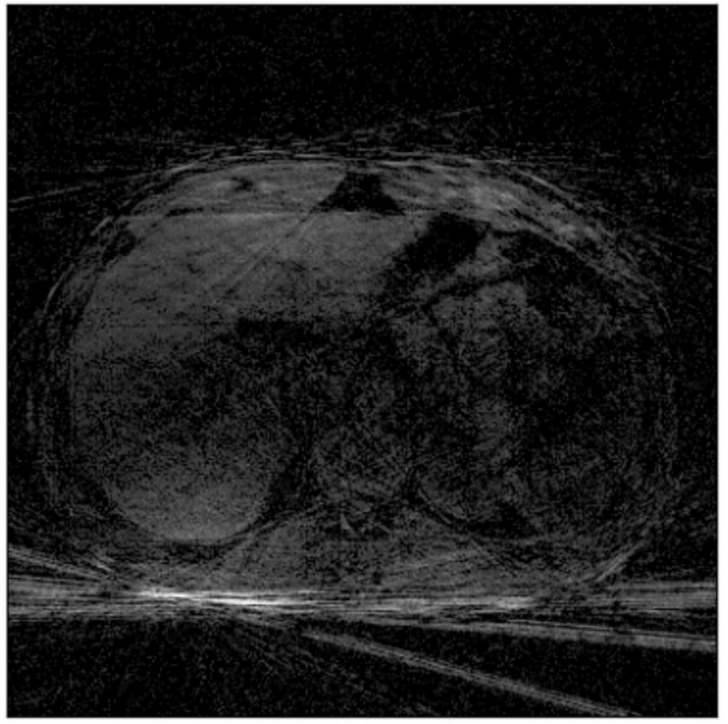
✓ 0.5s

(details in online repository)

Initial estimate (with density compensation)

$$x_{init} = F'S'Wb$$

```python
# create the operators
data_op = SenseNufftOp(sensitivity_maps, ktraj)

# initial estimate
with torch.no_grad():
    orig_est = data_op.adjoint(dcomp * kdata) / torch.sum(
        sensitivity_maps.abs() ** 2, dim=1, keepdim=True
    )
```



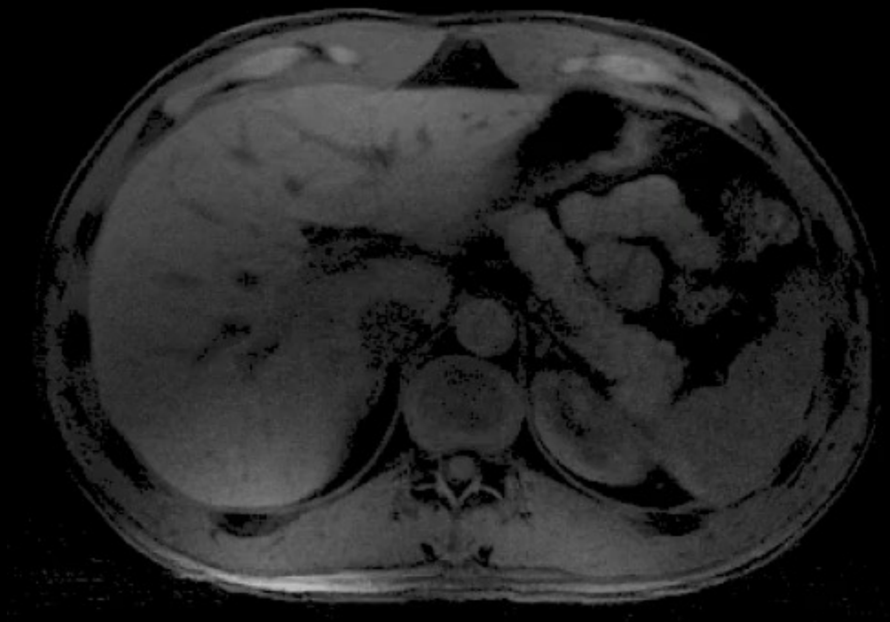
Step 6: Run compressed sensing reconstruction

Run compressed sensing reconstruction

```
# create the optimizer
opt = PrimalDualL1(
    data_operator=data_op,
    data_bound=sense_eig,
    reg_operator=reg_op,
    reg_bound=reg_eig,
    num_iterations=8,
    data_weights=dcomp,
)

# optimize!
with torch.no_grad():
    est = opt.solve(kdata, orig_est)
```

Run compressed sensing reconstruction

```
# create the optimizer
opt = PrimalDualL1(
    data_operator=data_op,
    data_bound=sense_eig,
    reg_operator=reg_op,
    reg_bound=reg_eig,
    num_iterations=8,
    data_weights=dcomp,
)

# optimize!
with torch.no_grad():
    est = opt.solve(kdata, orig_est)
```



References

GRASP data

1. Feng, L., Grimm, R., Block, K. T., Chandarana, H., Kim, S., Xu, J., ... & Otazo, R. (2014). Golden-angle radial sparse parallel MRI: combination of compressed sensing, parallel imaging, and golden-angle radial sampling for fast and flexible dynamic volumetric MRI. *MRM*, 72(3), 707-717.

Reconstruction algorithms

2. (Non-linear CG) Feng, L., Grimm, R., Block, K. T., Chandarana, H., Kim, S., Xu, J., ... & Otazo, R. (2014). Golden-angle radial sparse parallel MRI: combination of compressed sensing, parallel imaging, and golden-angle radial sampling for fast and flexible dynamic volumetric MRI. *MRM*, 72(3), 707-717.
3. (ADMM/Lagrangian) Boyd, S. P., & Vandenberghe, L. (2004). *Convex optimization*. Cambridge university press.
4. (ADMM/Lagrangian) Yang, J., Zhang, Y., & Yin, W. (2010). A fast alternating direction method for TVL1-L2 signal reconstruction from partial Fourier data. *IEEE Journal of Selected Topics in Signal Processing*, 4(2), 288-297.
5. (Primal-dual/MFISTA) Beck, A., & Teboulle, M. (2009). Fast gradient-based algorithms for constrained total variation image denoising and deblurring problems. *IEEE-TIP*, 18(11), 2419-2434.

Operators

6. (CG-SENSE) Pruessmann, K. P., Weiger, M., Börnert, P., & Boesiger, P. (2001). Advances in sensitivity encoding with arbitrary k-space trajectories. *MRM*, 46(4), 638-651.
7. (NUFFT) Beatty, P. J., Nishimura, D. G., & Pauly, J. M. (2005). Rapid gridding reconstruction with a minimal oversampling ratio. *IEEE-TMI*, 24(6), 799-808.
8. (NUFFT) Fessler, J. A., & Sutton, B. P. (2003). Nonuniform fast Fourier transforms using min-max interpolation. *IEEE-TSP*, 51(2), 560-574.
9. (NUFFT) Barnett, A. H., Magland, J., & af Klinteberg, L. (2019). A parallel nonuniform fast Fourier transform library based on an “Exponential of semicircle" kernel. *SIAM Journal on Scientific Computing*, 41(5), C479-C504.
10. (Toeplitz NUFFT) Feichtinger, H. G., Grochenig, K., & Strohmer, T. (1995). Efficient numerical methods in non-uniform sampling theory. *Numerische Mathematik*, 69, 423-440.

Partial list of reconstruction software packages

General reconstruction packages

- Gadgetron (C++, http://gadgetron.github.io/)
- BART (C/C++, https://mrirecon.github.io/bart/)
- sigpy (Python, https://sigpy.readthedocs.io/en/latest/)
- MIRT (Matlab and Julia, https://github.com/JeffFessler/mirt)
- MRIReco (Julia, https://github.com/MagneticResonanceImaging/MRIReco.jl)

Deep learning

- fastMRI (https://github.com/facebookresearch/fastMRI)
- DIRECT (https://github.com/NKI-AI/direct)

NUFFT Packages

- FINUFFT (C++, Python, https://github.com/flatironinstitute/finufft)
- MIRT NUFFT (Matlab, https://github.com/JeffFessler/mirt)
- NFFT (Julia, https://github.com/JuliaMath/NFFT.jl)
- sigpy NUFFT (Python, https://sigpy.readthedocs.io/en/latest/)
- gpuNUFFT (Python, Matlab, https://github.com/andyschwarzl/gpuNUFFT)
- PyNUFFT (Python, https://github.com/jyhmiinlin/pynufft)
- torchkbnufft (Python PyTorch, https://github.com/mmuckley/torchkbnufft)
- TF-KBNUFT (Python Tensorflow, https://github.com/zaccharieramzi/tfkbnufft)

Thank you!