pygrappa

Jun 19, 2020

Contents

CHAPTER 1

About

GRAPPA is a popular parallel imaging reconstruction algorithm. Unfortunately there aren't a lot of easy to use Python implementations of it or its many variants available, so I decided to release this simple package.

There are also a couple reference SENSE-like implementations that have made their way into the package. This is to be expected – a lot of later parallel imaging algorithms have hints of both GRAPPA- and SENSE-like inspirations.

1.1 Installation

1.1.1 Installation

Windows 10 Installation

If you are using Windows, then, first of all: sorry. This is not ideal, but I understand that it might not be your fault. I will assume you are trying to get pygrappa installed on Windows 10. I will further assume that you are using Python 3.7 64-bit build. We will need a C++ compiler to install pygrappa, so officially you should have "Microsoft Visual C++ Build Tools" installed. I haven't tried this, but it should work with VS build tools installed.

However, if you are not able to install the build tools, we can do it using the MinGW compiler instead. It'll be a little more involved than a simple *pip install*, but that's what you get for choosing Windows.

Steps:

- Download 64-bit fork of MinGW from
  https://sourceforge.net/projects/mingw-w64/
- Follow this guide:
  https://github.com/orlp/dev-on-windows/wiki/Installing-GCC–&-MSYS2
- Now you should be able to use gcc/g++/etc. from CMD-line
- Modify cygwinccompiler.py similar to
  https://github.com/tgalal/yowsup/issues/2494#issuecomment-388439162
  but using the version number *1916*:

```
def get_msvcr():
    """Include the appropriate MSVC runtime library if Python
    was built with MSVC 7.0 or later.
    """
    msc_pos = sys.version.find('MSC v.')
    if msc_pos != -1:
        msc_ver = sys.version[msc_pos+6:msc_pos+10]
        if msc_ver == '1300':
            # MSVC 7.0
            return ['msvcr70']
        elif msc_ver == '1310':
            # MSVC 7.1
            return ['msvcr71']
        elif msc_ver == '1400':
            # VS2005 / MSVC 8.0
            return ['msvcr80']
        elif msc_ver == '1500':
            # VS2008 / MSVC 9.0
            return ['msvcr90']
        elif msc_ver == '1600':
            # VS2010 / MSVC 10.0
            return ['msvcr100']
        elif msc_ver == '1916': # <- ADD THIS CONDITION
            # Visual Studio 2015 / Visual C++ 14.0
            return ['vcruntime140']
        else:
            raise ValueError(
                "Unknown MS Compiler version %s " % msc_ver)
```

- now run the command:

```
pip install --global-option build_ext --global-option \
    --compiler=mingw32 --global-option -DMS_WIN64 pygrappa
```

Hopefully this works for you. Refer to https://github.com/mckib2/pygrappa/issues/17 for a more detailed discussion.

This package is developed in Ubuntu 18.04 using Python 3.6.8. That's not to say it won't work on other things. You should submit an issue when it doesn't work like it says it should. The whole idea was to have an easy to use, pip-install-able GRAPPA module, so let's try to do that.

In general, it's a good idea to work inside virtual environments. I create and activate mine like this:

```
python3 -m venv /venvs/pygrappa
source /venvs/pygrappa/bin/activate
```

More information can be found in the venv documentation.

Installation under a Unix-based platform should then be as easy as:

```
pip install pygrappa
```

You will need a C/C++ compiler that supports the C++14 standard. See *Windows 10 Installation* for more info on installing under Windows.

```
pip install pygrappa
```

There are C/C++ extensions to be compiled, so you will need a compiler that supports either the C++11 or C++14 standard. See *Installation* for more instructions.

1.2 API Reference

Note: The upcoming *1.0.0* release will make changes to the API and simplify the interface considerably. The plans are to collect all GRAPPA-like methods and SENSE-like methods in their own interfaces:

```
pygrappa.grappa(
    kspace, calib=None, kernel_size=None,
    method='grappa', coil_axis=-1, options=None)
    pygrappa.sense(kspace, sens, coil_axis=-1, options)
```

The *method* parameter will allow the *grappa* interface to call the existing methods such as *tgrappa*, *mdgrappa*, etc. under the hood. The dictionary *options* can be used to pass in method-specific parameters. The SENSE interface will behave similarly.

The gridding interface is still an open question.

Progress on the *1.0.0* release can be found here

1.2.1 API Reference

pygrappa.grappa

Python GRAPPA implementation.

More efficient Python implementation of GRAPPA.

Notes

view_as_windows uses numpy.lib.stride_tricks.as_strided which may use up a lot of memory. This is more efficient as we get all the patches in one go as opposed to looping over the image in multiple dimensions. These are stored in temporary memmaps so we don't crash anyone's computer (from memory usage, at least...). Note that the recon is always stored in a temporary file memmap to begin with, since its initial size is zero-padded. The final output array or memmap is then initialized at the end with the correct size. This is because it's hard to resize memmaps, it's easier to create a temporary one and then copy over the contents to the final one.

We are looping over unique sampling patterns, similar to Miki Lustig's key-lookup table for kernels. It might be nice to train multiple kernel geometries simultaneously if possible, or at least have an option to do chunks at a time.

Currently each hole in kspace is being looped over when applying weights for a single kernel type. It would be nice to apply the weights for all corresponding holes simultaneously.

pygrappa.grappa.**grappa**(*kspace*, *calib*, *kernel_size=(5, 5)*, *coil_axis=-1*, *lamda=0.01*, *memmap=False*, *memmap_filename='out.memmap'*, *silent=True*)

GeneRalized Autocalibrating Partially Parallel Acquisitions.

Parameters

- **kspace** (*array_like*) – 2D multi-coil k-space data to reconstruct from. Make sure that the missing entries have exact zeros in them.
- **calib** (*array_like*) – Calibration data (fully sampled k-space).
- **kernel_size** (*tuple, optional*) – Size of the 2D GRAPPA kernel (kx, ky).
- **coil_axis** (*int, optional*) – Dimension holding coil data. The other two dimensions should be image size: (sx, sy).

- **lamda** (*float, optional*) – Tikhonov regularization for the kernel calibration.
- **memmap** (*bool, optional*) – Store data in Numpy memmaps. Use when datasets are too large to store in memory.
- **memmap_filename** (*str, optional*) – Name of memmap to store results in. File is only saved if memmap=True.
- **silent** (*bool, optional*) – Suppress messages to user.

Returns **res** – k-space data where missing entries have been filled in.

Return type array_like

Notes

Based on implementation of the GRAPPA algorithm[1] for 2D images.

If memmap=True, the results will be written to memmap_filename and nothing is returned from the function.

References

pygrappa.cgrappa

Used by autodoc_mock_imports.

cgrappa is Cython implementation of GRAPPA. It is faster than its Python counterparts, but is known to have bugs. It is probably due for a rewrite in the style of *mdgrappa*.

pygrappa.mdgrappa

Python implementation of multidimensional GRAPPA.

pygrappa.mdgrappa.**mdgrappa**(*kspace*, *calib=None*, *kernel_size=None*, *coil_axis=-1*, *lamda=0.01*, *nnz=None*, *weights=None*, *ret_weights=False*)

GeneRalized Autocalibrating Partially Parallel Acquisitions.

Parameters

- **kspace** (*N-D array*) – Measured undersampled complex k-space data. N-1 dimensions hold spatial frequency axes (kx, ky, kz, etc.). 1 dimension holds coil images (*coil_axis*). The missing entries should have exactly 0.
- **calib** (*N-D array or None, optional*) – Fully sampled calibration data. If *None*, calibration data will be extracted from the largest possible hypercube with origin at the center of k-space.
- **kernel_size** (*tuple or None, optional*) – The size of the N-1 dimensional GRAPPA kernels: (kx, ky, . . .). Default: (5,)*(kspace.ndim-1)
- **coil_axis** (*int, optional*) – Dimension holding coil images.
- **lamda** (*float, optional*) – Tikhonov regularization constant for kernel calibration.
- **nnz** (*int or None, optional*) – Number of nonzero elements in a multidimensional patch required to train/apply a kernel. Default: *sqrt(prod(kernel_size))*.
- **weights** (*dict, optional*) – Maps sampling patterns to trained kernels.

[1] Griswold, Mark A., et al. "Generalized autocalibrating partially parallel acquisitions (GRAPPA)." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 47.6 (2002): 1202-1210.

- **ret_weights** (*bool, optional*) – Return the trained weights as a dictionary mapping sampling patterns to kernels. Default is `False`.

Returns

- **res** (*array_like*) – k-space data where missing entries have been filled in.
- **weights** (*dict, optional*) – Returned if `ret_weights=True`.

Notes

Based on the GRAPPA algorithm described in[1].

All axes (except coil axis) are used for GRAPPA reconstruction.

References

pygrappa.igrappa

Python implementation of the iGRAPPA algorithm.

`pygrappa.igrappa.`**igrappa**(*kspace*, *calib*, *kernel_size=(5, 5)*, *k=0.3*, *coil_axis=-1*, *lamda=0.01*, *ref=None*, *niter=10*, *silent=True*, *backend=<function mdgrappa>*)

Iterative GRAPPA.

Parameters

- **kspace** (*array_like*) – 2D multi-coil k-space data to reconstruct from. Make sure that the missing entries have exact zeros in them.
- **calib** (*array_like*) – Calibration data (fully sampled k-space).
- **kernel_size** (*tuple, optional*) – Size of the 2D GRAPPA kernel (kx, ky).
- **k** (*float, optional*) – Regularization parameter for iterative reconstruction. Must be in the interval (0, 1).
- **coil_axis** (*int, optional*) – Dimension holding coil data. The other two dimensions should be image size: (sx, sy).
- **lamda** (*float, optional*) – Tikhonov regularization for the kernel calibration.
- **ref** (*array_like or None, optional*) – Reference k-space data. This is the true data that we are attempting to reconstruct. If provided, MSE at each iteration will be returned. If None, only reconstructed kspace is returned.
- **niter** (*int, optional*) – Number of iterations.
- **silent** (*bool, optional*) – Suppress messages to user.
- **backend** (*callable*) – GRAPPA function to use during each iteration. Default is `pygrappa.mdgrappa`.

Returns

- **res** (*array_like*) – k-space data where missing entries have been filled in.
- **mse** (*array_like, optional*) – MSE at each iteration. Returned if ref not None.

Raises `AssertionError` – If regularization parameter k is not in the interval (0, 1).

[1] Griswold, Mark A., et al. "Generalized autocalibrating partially parallel acquisitions (GRAPPA)." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 47.6 (2002): 1202-1210.

Notes

More or less implements the iterative algorithm described in [1].

References

pygrappa.hpgrappa

Python implementation of hp-GRAPPA.

pygrappa.hpgrappa.**hpgrappa**(*kspace*, *calib*, *fov*, *kernel_size=(5, 5)*, *w=None*, *c=None*, *ret_filter=False*, *coil_axis=-1*, *lamda=0.01*, *silent=True*)

High-pass GRAPPA.

Parameters

- **fov** (*tuple, (FOV_x, FOV_y)*) – Field of view (in m).
- **w** (*float, optional*) – Filter parameter: determines the smoothness of the filter boundary.
- **c** (*float, optional*) – Filter parameter: sets the cutoff frequency.
- **ret_filter** (*bool, optional*) – Returns the high pass filter determined by (w, c).

Notes

If w and/or c are None, then the closest values listed in Table 1 from[1] will be used.

F2 described by Equation [2] in[1] is used to generate the high pass filter.

References

pygrappa.seggrappa

Python implementation of the Segmented GRAPPA algorithm.

pygrappa.seggrappa.**seggrappa**(*kspace*, *calibs*, **args*, ***kwargs*)

Segmented GRAPPA.

See pygrappa.grappa() for full list of arguments.

Parameters **calibs** (*list of array_like*) – List of calibration regions.

Notes

A generalized implementation of the method described in[1]. Multiple ACS regions can be supplied to function. GRAPPA is run for each ACS region and then averaged to produce the final reconstruction.

[1] Huang, Feng, et al. "High-pass GRAPPA: An image support reduction technique for improved partially parallel imaging." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 59.3 (2008): 642-649.

[1] Park, Jaeseok, et al. "Artifact and noise suppression in GRAPPA imaging using improved k-space coil calibration and variable density sampling." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 53.1 (2005): 186-193.

References

pygrappa.tgrappa

TGRAPPA implementation.

pygrappa.tgrappa.**tgrappa**(*kspace*, *calib_size=(20, 20)*, *kernel_size=(5, 5)*, *coil_axis=-2*, *time_axis=-1*)

Temporal GRAPPA.

Parameters

- **kspace** (*array_like*) – 2+1D multi-coil k-space data to reconstruct from (total of 4 dimensions). Missing entries should have exact zeros in them.
- **calib_size** (*array_like, optional*) – Size of calibration region at the center of kspace.
- **kernel_size** (*tuple, optional*) – Desired shape of the in-plane calibration regions: (kx, ky).
- **coil_axis** (*int, optional*) – Dimension holding coil data.
- **time_axis** (*int, optional*) – Dimension holding time data.

Returns **res** – Reconstructed k-space data.

Return type array_like

Raises ValueError – When no complete ACS region can be found.

Notes

Implementation of the method proposed in[1].

The idea is to form ACS regions using data from adjacent time frames. For example, in the case of 1D undersampling using undersampling factor R, at least R time frames must be merged to form a completely sampled ACS. Then we can simply supply the undersampled data and the synthesized ACS to GRAPPA. Thus the heavy lifting of this function will be in determining the ACS calibration region at each time frame.

References

pygrappa.slicegrappa

Python implementation of the Slice-GRAPPA algorithm.

pygrappa.slicegrappa.**slicegrappa**(*kspace*, *calib*, *kernel_size=(5, 5)*, *prior='sim'*, *coil_axis=-2*, *time_axis=-1*, *slice_axis=-1*, *lamda=0.01*, *split=False*)

(Split)-Slice-GRAPPA for SMS reconstruction.

Parameters

- **kspace** (*array_like*) – Time frames of sum of k-space coil measurements for multiple slices.
- **calib** (*array_like*) – Single slice measurements for each slice present in kspace. Should be the same dimensions.
- **kernel_size** (*tuple, optional*) – Size of the GRAPPA kernel: (kx, ky).

[1] Breuer, Felix A., et al. "Dynamic autocalibrated parallel imaging using temporal GRAPPA (TGRAPPA)." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 53.4 (2005): 981-985.

- **prior** (*{ 'sim', 'kspace' }, optional*) – How to construct GRAPPA sources. GRAPPA weights are found by solving the least squares problem T = S W, where T are the targets (calib), S are the sources, and W are the weights. The possible options are:
  - 'sim': simulate SMS acquisition from calibration data, i.e., sources S = sum(calib, axis=slice_axis). This presupposes that the spatial locations of the slices in the calibration data are the same as in the overlapped kspace data. This is similar to how the k-t BLAST Wiener filter is constructed (see equation 1 in[2]).
  - 'kspace': uses the first time frame of the overlapped data as sources, i.e., S = kspace[1st time frame].

  This option is not used for Split-Slice-GRAPPA.
- **coil_axis** (*int, optional*) – Dimension that holds the coil data.
- **time_axis** (*int, optional*) – Dimension of kspace that holds the time data.
- **slice_axis** (*int, optional*) – Dimension of calib that holds the slice information.
- **lamda** (*float, optional*) – Tikhonov regularization for the kernel calibration.
- **split** (*bool, optional*) – Uses Split-Slice-GRAPPA kernel training method.

Returns **res** – Reconstructed slices for each time frame. res will always return the data in fixed order or shape: (nx, ny, num_coils, num_time_frames, num_slices).

Return type array_like

Raises `NotImplementedError` – When "prior" is an invalid option.

Notes

This function implements both the Slice-GRAPPA algorithm as described in[1] and the Split-Slice-GRAPPA algorithm as first described in[3].

References

pygrappa.splitslicegrappa

Python implementation of Split-Slice-GRAPPA.

`pygrappa.splitslicegrappa.`**splitslicegrappa**(**args*, ***kwargs*)

Split-Slice-GRAPPA.

Notes

This is an alias for pygrappa.slicegrappa(split=True). See pygrappa.slicegrappa() for more information.

[2] Sigfridsson, Andreas, et al. "Improving temporal fidelity in k-t BLAST MRI reconstruction." International Conference on Medical Image Computing and Computer-Assisted Intervention. Springer, Berlin, Heidelberg, 2007.

[1] Setsompop, Kawin, et al. "Blipped-controlled aliasing in parallel imaging for simultaneous multislice echo planar imaging with reduced g-factor penalty." Magnetic resonance in medicine 67.5 (2012): 1210-1224.

[3] Cauley, Stephen F., et al. "Interslice leakage artifact reduction technique for simultaneous multislice acquisitions." Magnetic resonance in medicine 72.1 (2014): 93-102.

pygrappa.grappaop

Python implementation of the GRAPPA operator formalism.

pygrappa.grappaop.**grappaop**(*calib*, *coil_axis=-1*, *lamda=0.01*)

GRAPPA operator for Cartesian calibration datasets.

Parameters

- **calib** (*array_like*) – Calibration region data. Usually a small portion from the center of kspace.
- **coil_axis** (*int, optional*) – Dimension holding coil data.
- **lamda** (*float, optional*) – Tikhonov regularization parameter. Set to 0 for no regularization.

Returns **Gx, Gy** – GRAPPA operators for both the x and y directions.

Return type array_like

Notes

Produces the unit operator described in[1].

This seems to only work well when coil sensitivities are very well separated/distinct. If coil sensitivities are similar, operators perform poorly.

References

pygrappa.radialgrappaop

Python implementation of Radial GRAPPA operator.

pygrappa.radialgrappaop.**radialgrappaop**(*kx*, *ky*, *k*, *nspokes=None*, *spoke_axis=-2*, *coil_axis=-1*, *spoke_axis_coord=-1*, *lamda=0.01*, *ret_lGtheta=False*, *traj_warn=True*)

Non-Cartesian Radial GRAPPA operator.

Parameters

- **ky** (*kx,*) – k-space coordinates of kspace data, k. kx and ky are 2D arrays containing (sx, nr) : (number of samples along ray, number of rays).
- **k** (*array_like*) – Complex kspace data corresponding to the measurements at locations kx, ky. k has three dimensions: sx, nr, and coil.
- **nspokes** (*int, optional*) – Number of spokes. Used when (kx, ky) and k are given with flattened sample and spoke axes, i.e., (sx*nr, nc).
- **spoke_axis** (*int, optional*) – Axis of k that contains the spoke data. Not for kx, ky: see spoke_axis_coord to specify spoke axis for kx and ky.
- **coil_axis** (*int, optional*) – Axis of k that contains the coil data.
- **spoke_axis_coord** (*int, optional*) – Axis of kx and ky that hold the spoke data.
- **lamda** (*float, optional*) – Tikhonov regularization term used both for fitting Gtheta and log(Gx), log(Gy).

[1] Griswold, Mark A., et al. "Parallel magnetic resonance imaging using the GRAPPA operator formalism." Magnetic resonance in medicine 54.6 (2005): 1553-1556.

- **ret_lGtheta** (*bool, optional*) – Return log(Gtheta) instead of Gx, Gy.
- **traj_warn** (*bool, optional*) – Warn about potential inconsistencies in trajectory, e.g., not shaped correctly.

Returns **Gx, Gy** – GRAPPA operators along the x and y axes.

Return type array_like

Raises `AssertionError` – If kx and ky do not have spokes along spoke_axis_coord or if the standard deviation of distance between spoke points is greater than or equal to 1e-10.

Notes

Implements the radial training scheme for self calibrating GRAPPA operators in[1]. Too many coils could lead to instability of matrix exponents and logarithms – use PCA or other suitable coil combination technique to reduce dimensionality if needed.

References

pygrappa.ttgrappa

Python implementation of through-time GRAPPA.

`pygrappa.ttgrappa.`**ttgrappa**(*kx*, *ky*, *kspace*, *cx*, *cy*, *calib*, *kernel_size=25*, *kernel_radius=None*, *max_kernel_size=25*, *coil_axis=-1*, *time_axis=-2*, *lamda=0.01*)

Through-time GRAPPA.

Parameters

- **ky** (*kx,*) – k-space coordinates of kspace data, kspace. kx and ky are 1D arrays.
- **kspace** (*array_like*) – Complex kspace data corresponding to the measurements at locations kx, ky. kspace has two dimensions: data and coil. Unsampled points should be exactly 0.
- **cy** (*cx,*) – k-space coordinates of calibration kspace data. cx and cy are 1D arrays.
- **calib** (*array_like*) – Complex kspace data corresponding to the measurements at locations cx, cy. calib has three dimensions: data, time, and coil.
- **kernel_size** (*int, optional*) – Number of points to use as sources for kernel training. This many nearest neighbors to the targets will be chosen.
- **kernel_radius** (*float, optional*) – If not None, this radius will be used instead of kernel_size. All sources within this radius of the target will be chosen. Has units same as kx, ky.
- **max_kernel_size** (*int, optional*) – Maximum number of points in ball when using kernel_radius. If more sources are found, then randomly choose max_kernel_size of them.
- **coil_axis** (*int, optional*) – Dimension of kspace and calib holding coil data.
- **time_axis** (*int, optional*) – Dimension of calib holding time data.
- **lamda** (*float, optional*) – Tikhonov regularization for the kernel calibration.

Returns **res** – The reconstructed measurements with the same size as kspace.

[1] Seiberlich, Nicole, et al. "Self-calibrating GRAPPA operator gridding for radial and spiral trajectories." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 59.4 (2008): 930-935.

Return type array_like

Notes

Implements the through-time GRAPPA algorithm for non-Cartesian reconstruction as described in[1].

This implementation uses a kd-tree for kernel selection similar to[2]. This simplifies searches for kernel geometries and helps make this implementation trajectory agnostic.

References

pygrappa.pars

Python implementation of the PARS algorithm.

pygrappa.pars.**pars**(*kx*, *ky*, *k*, *sens*, *tx=None*, *ty=None*, *kernel_size=25*, *kernel_radius=None*, *coil_axis=-1*)

Parallel MRI with adaptive radius in k-space.

Parameters

- **ky** (*kx,*) – Sample points in kspace corresponding to measurements k. kx, kx are 1D arrays.
- **k** (*array_like*) – Complex kspace coil measurements corresponding to points (kx, ky).
- **sens** (*array_like*) – Coil sensitivity maps with shape of desired reconstruction.
- **ty** (*tx,*) – Sample points in kspace defining the grid of ifft2(sens). If None, then tx, ty will be generated from a meshgrid with endpoints [min(kx), max(kx), min(ky), max(ky)].
- **kernel_size** (*int, optional*) – Number of nearest neighbors to use when interpolating kspace.
- **kernel_radius** (*float, optional*) – Raidus in kspace (units same as (kx, ky)) to select neighbors when training kernels.
- **coil_axis** (*int, optional*) – Dimension holding coil data.

Returns **res** – Reconstructed image space on a Cartesian grid with the same shape as sens.

Return type array_like

Notes

Implements the algorithm described in[1].

Using kernel_radius seems to perform better than kernel_size.

[1] Seiberlich, Nicole, et al. "Improved radial GRAPPA calibration for real-time free-breathing cardiac imaging." Magnetic resonance in medicine 65.2 (2011): 492-505.

[2] Luo, Tianrui, et al. "A GRAPPA algorithm for arbitrary 2D/3D non-Cartesian sampling trajectories with rapid calibration." Magnetic resonance in medicine 82.3 (2019): 1101-1112.

[1] Yeh, Ernest N., et al. "3Parallel magnetic resonance imaging with adaptive radius in k-space (PARS): Constrained image reconstruction using k-space locality in radiofrequency coil encoded data." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 53.6 (2005): 1383-1392.

References

pygrappa.grog

Python implmentation of the GROG algorithm.

pygrappa.grog.**grog**(*kx*, *ky*, *k*, *N*, *M*, *Gx*, *Gy*, *precision=2*, *radius=0.75*, *Dx=None*, *Dy=None*, *coil_axis=-1*, *ret_image=False*, *ret_dicts=False*, *use_primefac=False*, *remove_os=True*, *inverse=False*)

GRAPPA operator gridding.

Parameters

- **ky** (*kx,*) – k-space coordinates (kx, ky) of measured data k. kx, ky should each be a 1D array. Must both be either float or double.
- **k** (*array_like*) – Measured k-space data at points (kx, ky).
- **M** (*N,*) – Desired resolution of Cartesian grid.
- **Gy** (*Gx,*) – Unit GRAPPA operators.
- **precision** (*int, optional*) – Number of decimal places to round fractional matrix powers to.
- **radius** (*float, optional*) – Radius of ball in k-space to from Cartesian targets from which to select source points.
- **Dy** (*Dx,*) – Dictionaries of precomputed fractional matrix powers.
- **coil_axis** (*int, optional*) – Axis holding coil data.
- **ret_image** (*bool, optional*) – Return image space result instead of k-space.
- **ret_dicts** (*bool, optional*) – Return dictionaries of fractional matrix powers.
- **use_primefac** (*bool, optional*) – Use prime factorization to speed-up fractional matrix power precomputations.
- **remove_os** (*bool, optional*) – Remove oversampling factor.
- **inverse** (*bool, optional*) – Do the inverse gridding operation, i.e., Cartesian points to (kx, ky).

Returns

- **res** (*array_like*) – Cartesian gridded k-space (or image).
- **Dx, Dy** (*dict, optional*) – Fractional matrix power dictionary for both Gx and Gy.

Raises

- AssertionError – When (kx, ky) have different types.
- AssertionError – When (kx, ky) and k do not have matching types, i.e., if (kx, ky) are float32, k must be complex64.

Notes

Implements the GROG algorithm as described in[1].

[1] Seiberlich, Nicole, et al. "Self-calibrating GRAPPA operator gridding for radial and spiral trajectories." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 59.4 (2008): 930-935.

References

pygrappa.nlgrappa_matlab

Python port of MATLAB script.

pygrappa.nlgrappa_matlab.**nlgrappa_matlab**(*reduced_fourier_data*, *ORF*, *pe_loc*, *acs_data*, *acs_line_loc*, *num_block*, *num_column*, *times_comp*)

Python port of original NL-GRAPPA script.

Parameters

- **reduced_fourier_data** (*array_like*) – undersampled k-space data
- **ORF** (*int*) – outer reduction factor
- **pe_loc** (*array_like*) – undersampled phase-encoding lines' location
- **acs_data** (*array_like*) – auto-calibration signal data (middle region of k-space)
- **acs_line_loc** (*array_like*) – auto-calibration signal lines' location
- **num_block** (*int*) – number of blocks
- **num_column** (*int*) – number of columns
- **times_comp** (*int*) – times of the number of the first-order terms (the number of the second-order terms = time_comp X the number of the first-order terms)

Returns

- **full_fourier_data** (*array_like*) – reconstructed k-space (with ACS replacement)
- **rec_img** (*array_like*) – reconstructed image
- **coef0** (*array_like*) – coefficients for reconstruction

Notes

time_comp parameter As the parameter time_comp increases, relevant second-order terms are added for reconstruction. When time_comp = 1

MATLAB script: Written by: Yuchou Chang, University of Wisconsin - Milwaukee Email: yuchou@uwm.edu; leiying@uwm.edu Created on Oct. 12, 2011

References

pygrappa.gfactor

Calculate g-factor maps.

pygrappa.gfactor.**gfactor**(*coils*, *Rx*, *Ry*, *coil_axis=-1*, *tol=1e-06*)

Compute g-factor map for coil sensitities and accelerations.

Parameters

- **C** (*array_like*) – Array of coil sensitivities
- **Ry** (*int*) – x acceleration
- **Ry** – y acceleration

- **coil_axis** (*int, optional*) – Dimension holding coil data.
- **tol** (*float, optional*) –

Returns **g** – g-factor map

Return type array_like

Notes

Adapted from John Pauly's MATLAB script found at[1].

References

pygrappa.gfactor.**gfactor_single_coil_R2**(*coil*, *Rx=2*, *Ry=1*)

Specific example of a single homogeneous coil, R=2.

Parameters

- **coil** (*array_like*) – Single coil sensitivity.
- **Ry** (*int*) – x acceleration
- **Ry** – y acceleration

Returns **g** – g-factor map

Return type array_like

Notes

Analytical solution for a single, homogeneous coil with an undersampling factor of R=2. Equation 11 in[2].

Comparing head-to-head with pygrappa.gfactor(), this does produce different results. I don't know which one is more correct...

References

pygrappa.sense1d

Python implementation of SENSE.

pygrappa.sense1d.**sense1d**(*im*, *sens*, *Rx=1*, *Ry=1*, *coil_axis=-1*, *imspace=True*)

Sensitivity Encoding for Fast MRI (SENSE) along one dimension.

Parameters

- **im** (*array_like*) – Array of the aliased 2D multicoil coil image. If imspace=False, im is the undersampled k-space data.
- **sens** (*array_like*) – Complex coil sensitivity maps with the same dimensions as im.
- **Ry** (*Rx,*) – Acceleration factor in x and y. One of Rx, Ry must be 1. If both are 1, then this is Roemer's optimal coil combination.
- **coil_axis** (*int, optional*) – Dimension holding coil data.

[1] https://web.stanford.edu/class/ee369c/restricted/ Solutions/assignment_4_solns.pdf

[2] Blaimer, Martin, et al. "Virtual coil concept for improved parallel MRI employing conjugate symmetric signals." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 61.1 (2009): 93-102.

- **imspace** (*bool, optional*) – If im is image space or k-space data.

Returns **res** – Unwrapped single coil reconstruction.

Return type array_like

Notes

Implements the algorithm first described in[1]. This implementation is based on the MATLAB tutorial found in[2].

This implementation handles only regular undersampling along a single dimension. Arbitrary undersampling is not supported by this function.

Odd Rx, Ry seem to behave strangely, i.e. not as well as even factors. Right now I'm padding im and sens by 1 and removing at end.

References

pygrappa.cgsense

Python implementation of iterative and CG-SENSE.

pygrappa.cgsense.**cgsense**(*kspace*, *sens*, *coil_axis=-1*)

Conjugate Gradient SENSE for arbitrary Cartesian acquisitions.

Parameters

- **kspace** (*array_like*) – Undersampled kspace data with exactly 0 in place of missing samples.
- **sens** (*array_like*) – Coil sensitivity maps.
- **coil_axis** (*int, optional*) – Dimension of kspace and sens holding the coil data.

Returns **res** – Single coil unaliased estimate (imspace).

Return type array_like

Notes

Implements a Cartesian version of the iterative algorithm described in[1]. It can handle arbitrary undersampling of Cartesian acquisitions and arbitrarily-dimensional datasets. All dimensions except `coil_axis` will be used for reconstruction.

This implementation uses the scipy.sparse.linalg.cg() conjugate gradient algorithm to solve A^H A x = A^H b.

References

The exact API of all functions and classes, as given by the docstrings. The API documents expected types and allowed features for all functions, and all parameters available for the algorithms.

A full catalog can be found in the *API Reference* page.

[1] Pruessmann, Klaas P., et al. "SENSE: sensitivity encoding for fast MRI." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 42.5 (1999): 952-962.

[2] https://users.fmrib.ox.ac.uk/~mchiew/docs/ SENSE_tutorial.html

[1] Pruessmann, Klaas P., et al. "Advances in sensitivity encoding with arbitrary k-space trajectories." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 46.4 (2001): 638-651.

CHAPTER 2

Usage

2.1 Usage

Note: These should probably be moved into docstrings for each method.

pygrappa.grappa() implements GRAPPA ([1]) for arbitrarily sampled Cartesian datasets. It is called with undersampled k-space data and calibration data (usually a fully sampled portion of the center of k-space). The unsampled points in k-space should be exactly 0:

```
from pygrappa import grappa

# These next two lines are to show you the sizes of kspace and
# calib -- you need to bring your own data.  It doesn't matter
# where the coil dimension is, you just need to let 'grappa' know
# when you call it by providing the 'coil_axis' argument
sx, sy, ncoils = kspace.shape[:]
cx, cy, ncoils = calib.shape[:]

# Here's the actual reconstruction
res = grappa(kspace, calib, kernel_size=(5, 5), coil_axis=-1)

# Here's the resulting shape of the reconstruction.  The coil
# axis will end up in the same place you provided it in
sx, sy, ncoils = res.shape[:]
```

If calibration data is in the k-space data, simply extract it (make sure to call the ndarray.copy() method, may break if using reference to the original k-space data):

```
from pygrappa import grappa

```

(continues on next page)

[1] Griswold, Mark A., et al. "Generalized autocalibrating partially parallel acquisitions (GRAPPA)." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 47.6 (2002): 1202-1210.

(continued from previous page)

```
sx, sy, ncoils = kspace.shape[:] # center 20 lines are ACS
ctr, pd = int(sy/2), 10
calib = kspace[:, ctr-pd:ctr+pad, :].copy() # call copy()!

# coil_axis=-1 is default, so if coil dimension is last we don't
# need to explicity provide it
res = grappa(kspace, calib, kernel_size=(5, 5))
sx, sy, ncoils = res.shape[:]
```

A very similar GRAPPA implementation with the same interface can be called like so:

```
from pygrappa import cgrappa
res = cgrappa(kspace, calib, kernel_size=(5, 5), coil_axis=-1)
```

This function uses much of the same code as the Python grappa() implementation, but has certain parts written in C++ and all compiled using Cython. It runs about twice as fast. It will probably become the default GRAPPA implementation in future releases.

vcgrappa() is a VC-GRAPPA ([2]) implementation that simply constructs conjugate virtual coils, appends them to the coil dimension, and passes everything through to *cgrappa()*. The function signature is identical to *pygrappa.grappa()*.

For reconstructions with more than 2 dimensions, there is a generalized multidimensional implementation called *mdgrappa()* that can be called as follows:

```
from pygrappa import mdgrappa
res = mdgrappa(kspace, calib, kernel_size=(5, 5, 5)) # e.g., 3D
```

igrappa() is an Iterative-GRAPPA ([3]) implementation that can be called as follows:

```
from pygrappa import igrappa
res = igrappa(kspace, calib, kernel_size=(5, 5))

# You can also provide the reference kspace to get the MSE at
# each iteration, showing you the performance.  Regularization
# parameter k (as described in paper) can also be provided:
res, mse = igrappa(kspace, calib, k=0.6, ref=ref_kspace)
```

igrappa() makes calls to *cgrappa()* on the back end.

hpgrappa() implements the High-Pass GRAPPA (hp-GRAPPA) algorithm ([4]). It requires FOV to construct an appropriate high pass filter. It can be called as:

```
from pygrappa import hpgrappa
res = hpgrappa(kspace, calib, fov=(FOV_x, FOV_y))
```

seggrappa() is a generalized Segmented GRAPPA implementation ([5]). It is supplied a list of calibration regions, *cgrappa* is run for each, and all the reconstructions are averaged together to yield the final image. It can be called with all the normal *cgrappa* arguments:

[2] Blaimer, Martin, et al. "Virtual coil concept for improved parallel MRI employing conjugate symmetric signals." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 61.1 (2009): 93-102.

[3] Zhao, Tiejun, and Xiaoping Hu. "Iterative GRAPPA (iGRAPPA) for improved parallel imaging reconstruction." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 59.4 (2008): 903-907.

[4] Huang, Feng, et al. "High-pass GRAPPA: An image support reduction technique for improved partially parallel imaging." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 59.3 (2008): 642-649.

[5] Park, Jaeseok, et al. "Artifact and noise suppression in GRAPPA imaging using improved k-space coil calibration and variable density sampling." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 53.1 (2005): 186-193.

```
from pygrappa import seggrappa

cx1, cy1, ncoil = calib1.shape[:]
cx2, cy2, ncoil = calib2.shape[:]
res = seggrappa(kspace, [calib1, calib2])
```

TGRAPPA is a Temporal GRAPPA implementation ([6]) and does not require calibration data. It can be called as:

```
from pygrappa import tgrappa

sx, sy, ncoils, nt = kspace.shape[:]
res = tgrappa(
    kspace, calib_size=(20, 20), kernel_size=(5, 5),
    coil_axis=-2, time_axis=-1)
```

Calibration region size and kernel size must be provided. The calibration regions will be constructed in a greedy manner: once enough time frames have been consumed to create an entire ACS, GRAPPA will be run. TGRAPPA uses the *cgrappa* implementation for its speed.

slicegrappa() is a Slice-GRAPPA ([7]) implementation that can be called like:

```
from pygrappa import slicegrappa

sx, sy, ncoils, nt = kspace.shape[:]
sx, sy, ncoils, sl = calib.shape[:]
res = slicegrappa(kspace, calib, kernel_size=(5, 5), prior='sim')
```

kspace is assumed to SMS-like with multiple collapsed slices and multiple time frames that each need to be separated. *calib* are the individual slices' kspace data at the same size/resolution. *prior* tells the Slice-GRAPPA algorithm how to construct the sources, that is, how to solve T = S W, where T are the targets (calibration data), S are the sources, and W are GRAPPA weights. *prior='sim'* creates S by simulating the SMS acquisition, i.e., S = sum(calib, slice_axis). *prior='kspace'* uses the first time frame from the *kspace* data, i.e., S = kspace[1st time frame]. The result is an array containing all target slices for all time frames in *kspace*.

Similarly, Split-Slice-GRAPPA ([8]) can be called like so:

```
from pygrappa import splitslicegrappa as ssgrappa

sx, sy, ncoils, nt = kspace.shape[:]
sx, sy, ncoils, sl = calib.shape[:]
res = ssgrappa(kspace, calib, kernel_size=(5, 5))

# Note that pygrappa.splitslicegrappa is an alias for
# pygrappa.slicegrappa(split=True), so it can also be called
# like this:
from pygrappa import slicegrappa
res = slicegrappa(kspace, calib, kernel_size=(5, 5), split=True)
```

grappaop returns two unit GRAPPA operators ([9],[10]) found from a 2D Cartesian calibration dataset:

[6] Breuer, Felix A., et al. "Dynamic autocalibrated parallel imaging using temporal GRAPPA (TGRAPPA)." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 53.4 (2005): 981-985.

[7] Setsompop, Kawin, et al. "Blipped-controlled aliasing in parallel imaging for simultaneous multislice echo planar imaging with reduced g-factor penalty." Magnetic resonance in medicine 67.5 (2012): 1210-1224.

[8] Cauley, Stephen F., et al. "Interslice leakage artifact reduction technique for simultaneous multislice acquisitions." Magnetic resonance in medicine 72.1 (2014): 93-102.

[9] Griswold, Mark A., et al. "Parallel magnetic resonance imaging using the GRAPPA operator formalism." Magnetic resonance in medicine 54.6 (2005): 1553-1556.

[10] Blaimer, Martin, et al. "2D-GRAPPA-operator for faster 3D parallel MRI." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 56.6 (2006): 1359-1364.

```
from pygrappa import grappaop

sx, sy, ncoils = calib.shape[:]
Gx, Gy = grappaop(calib, coil_axis=-1)
```

See the examples to see how to use the GRAPPA operators to reconstruct datasets.

Similarly, *radialgrappaop()* returns two unit GRAPPA operators[13] found from a radial calibration dataset:

```
from pygrappa import radialgrappaop
sx, nr = kx.shape[:] # sx: number of samples along each spoke
sx, nr = ky.shape[:] # nr: number of rays/spokes
sx, nr, nc = k.shape[:] # nc is number of coils

Gx, Gy = radialgrappaop(kx, ky, k)
```

For large number of coils, warnings will appear about matrix logarithms and exponents, but I think it should be fine.

ttgrappa implements the through-time GRAPPA algorithm ([11]). It accepts arbitrary k-space sampling locations and measurements along with corresponding fully sampled calibration data. The kernel is specified by the number of points desired, not a tuple as is usually the case:

```
from pygrappa import ttgrappa

# kx, ky are both 1D arrays describing the points (kx, ky)
# sampled in kspace.  kspace is a matrix with two dimensions:
# (meas., coil) corresponding to the measurements takes at each
# (kx, ky) from each coil.  (cx, cy) and calib are similarly
# supplied.  kernel_size is the number of nearest neighbors used
# for the least squares fit.  25 corresponds to a kernel size of
# (5, 5) for Cartesian GRAPPA:

res = ttgrappa(kx, ky, kspace, cx, cy, calib, kernel_size=25)
```

PARS[12] is an older parallel imaging algorithm, but it checks out. It can be called like so:

```
from pygrappa import pars

# Notice we provide the image domain coil sensitivity maps: sens
res = pars(kx, ky, kspace, sens, kernel_radius=.8, coil_axis=-1)

# You can use kernel_size instead of kernel_radius, but it seems
# that kernel_radius gives better reconstructions.
```

In general, PARS is slower in this Python implementation because the size of the kernels change from target point to target point, so we have to loop over every single one. Notice that *pars* returns the image domain reconstruction on the Cartesian grid, not interpolated k-space as most methods in this package do.

GROG[14] is called with trajectory information and unit GRAPPA operators Gx and Gy:

[13] Seiberlich, Nicole, et al. "Self-calibrating GRAPPA operator gridding for radial and spiral trajectories." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 59.4 (2008): 930-935.

[11] Seiberlich, Nicole, et al. "Improved radial GRAPPA calibration for real-time free-breathing cardiac imaging." Magnetic resonance in medicine 65.2 (2011): 492-505.

[12] Yeh, Ernest N., et al. "3Parallel magnetic resonance imaging with adaptive radius in k-space (PARS): Constrained image reconstruction using k-space locality in radiofrequency coil encoded data." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 53.6 (2005): 1383-1392.

[14] Seiberlich, Nicole, et al. "Self-calibrating GRAPPA operator gridding for radial and spiral trajectories." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 59.4 (2008): 930-935.

```
from pygrappa import grog

# (N, M) is the resolution of the desired Cartesian grid
res = grog(kx, ky, k, N, M, Gx, Gy)

# Precomputations of fractional matrix powers can be accelerated
# using a prime factorization technique submitted to ISMRM 2020:
res = grog(kx, ky, k, N, M, Gx, Gy, use_primefac=True)
```

See *examples.basic_radialgrappaop.py* for usage example.

Esoterically, forward and inverse gridding are supported out of the box with this implementation of GROG, i.e., non-Cartesian -> Cartesian can be reversed. It's not perfect and I've never heard of anyone doing this via GROG, but check out *examples.inverse_grog* for more info.

NL-GRAPPA uses machine learning feature augmentation to reduce model- based reconstruction error[15]. It's implementation is based on the original script, so its function signature looks different than normal. Please see example for better understanding of arguments. It can be called like so:

```
from pygrappa import nlgrappa_matlab
res = nlgrappa_matlab(
    kspace_u, R, pe_loc, calib, acs_line_loc, num_block,
    num_column, times_comp)
```

You might need to play around with the arguments to get good images. The implementation is pretty much a straight mapping of the original MATLAB script to Python, so performance is not going to be very good compared to the other GRAPPA implementations in this package.

There was Python implementation in previous versions of pygrappa, but it never worked correctly and raises an exception now if you try to call it.

g-factor maps show geometry factor and a general sense of how well parallel imaging techniques like GRAPPA will work. Coil sensitivities must be known for to use this function as well as integer acceleration factors in x and y:

```
from pygrappa import gfactor
g = gfactor(sens, Rx, Ry)
```

SENSE implements the algorithm described in[16] for unwrapping aliased images along a single axis. Coil sensitivity maps must be provided. Coil images may be provided in image domain or k-space with the approprite flag:

```
from pygrappa import sense1d
res = sense1d(im, sens, Rx=2, coil_axis=-1)

# Or, kspace data for coil images may be provided:
res = sense1d(kspace, sens, Rx=2, coil_axis=-1, imspace=False)
```

CG-SENSE implements a Cartesian version of the algorithm described in[17]. It works for arbitrary undersampling of Cartesian datasets. Undersampled k-space and coil sensitivity maps are provided:

```
from pygrappa import cgsense
res = cgsense(kspace, sens, coil_axis=-1)
```

[15] Chang, Yuchou, Dong Liang, and Leslie Ying. "Nonlinear GRAPPA: A kernel approach to parallel MRI reconstruction." Magnetic resonance in medicine 68.3 (2012): 730-740.

[16] Pruessmann, Klaas P., et al. "SENSE: sensitivity encoding for fast MRI." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 42.5 (1999): 952-962.

[17] Pruessmann, Klaas P., et al. "Advances in sensitivity encoding with arbitrary k-space trajectories." Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine 46.4 (2001): 638-651.

Although SENSE is more commonly known as an image domain parallel imaging reconstruction technique, it is useful to include in this package for comparison to kernel based and hybrid reconstructions.

2.2 References

See the *Usage* page. Also see the *examples* module. It has several scripts showing basic usage. Docstrings are also a great resource – check them out for all possible arguments and usage info.

You can run examples from the command line by calling them like this:

```
python -m pygrappa.examples.[example-name]

# For example, if I wanted to try out TGRAPPA:
python -m pygrappa.examples.basic_tgrappa
```

Python Module Index

p

Index