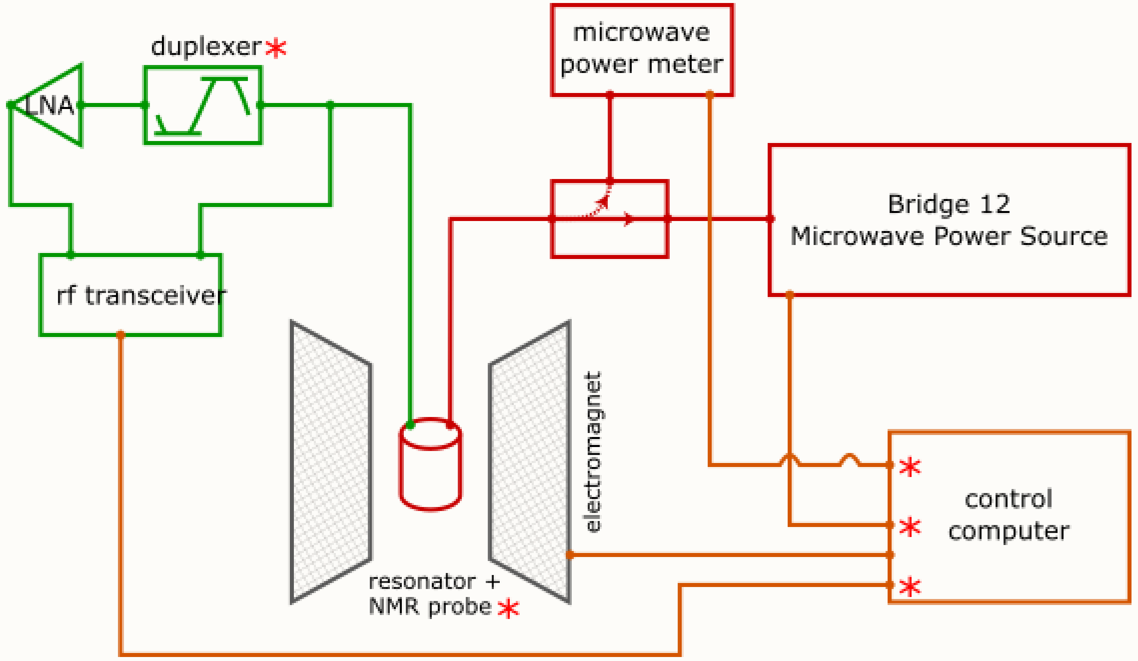
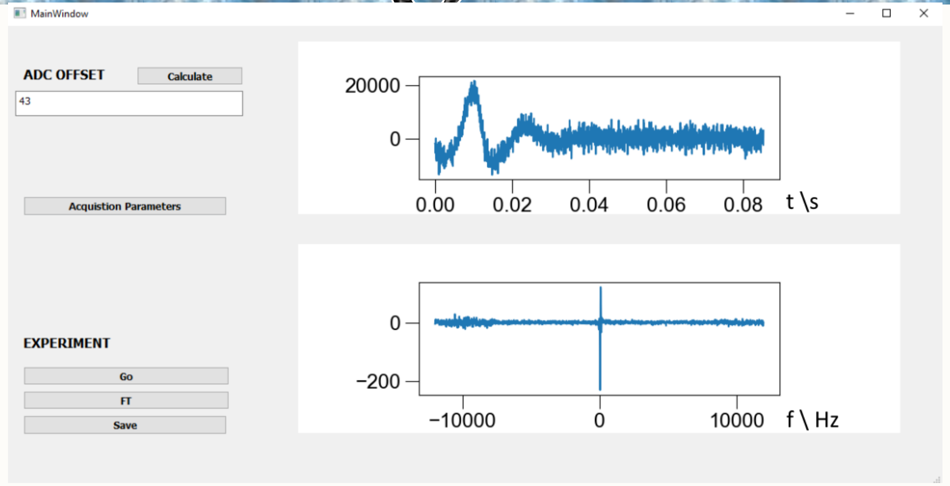
Table of Contents

Intro

Modular Instrument Design






```
print("\nLOADING PULSE PROG...\n")
if phase_cycling:
    SpinCore_pp.load([
        ('marker','start',1),
        ('phase_reset',1),
        ('delay_TTL',deblank),
        ('pulse_TTL',p90,'ph1',r_[0,1,2,3]),
        ('delay',tau),
        ('delay_TTL',deblank),
        ('pulse_TTL',2.0*p90,'ph2',r_[0,2]),
        ('delay',deadtime),
        ('acquire',acq_time),
        ('delay',repetition),
        ('jumpto','start')
        ])
```

pySpecData → our library (starting in Songi’s lab, carried to Freed lab, continuing with it at Syracuse) → http://github.com/jmfrancklab/pyspecdata

Some General Utilities

Uniform Code for Files on Different Computers

Designed for integration with different computers:

```
d = find_file('200212_IR_3_30dBm', exp_type='test_equip')
```

(I have configured a data directory on my computer, and there is a subdirectory called `test_equip` that lives inside this → look for a file named `200212_IR_3_30dBm` inside that subdirectory).

Automatic filetype recognition.

If file is missing, generates a message on how to download the data:

```
Traceback (most recent call last):
  File "fitdata_test.py", line 82, in
<module>
    directory = getDATADIR(exp_type =
'test_equip' ))
  File
"c:\users\johnf\notebook\pyspecdata\pyspecda
ta\core.py", line 6629, in __init__
    check_only=True, directory=directory)
  File
"c:\users\johnf\notebook\pyspecdata\pyspecda
ta\core.py", line 1042, in h5nodebypath
    +errmsg)
AttributeError: You're checking for a node
in a file (200212_IR_3_30dBm.h5) that does
not exist
I can't find 200212_IR_3_30dBm.h5 in
C:\Users\johnf\exp_data\test_equip\, so I'm
going to search for t in your rclone remotes
checking remote g_syr:
You should be able to retrieve this file
with:
rclone copy -v --include
'200212_IR_3_30dBm.h5'
g_syr:exp_data/test_equip
C:\\Users\\johnf\\exp_data\\test_equip
```

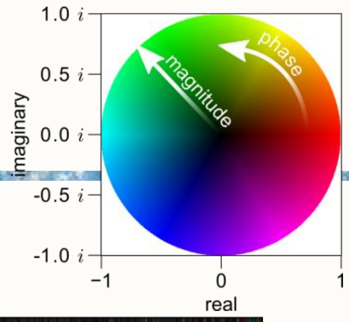
figure lists

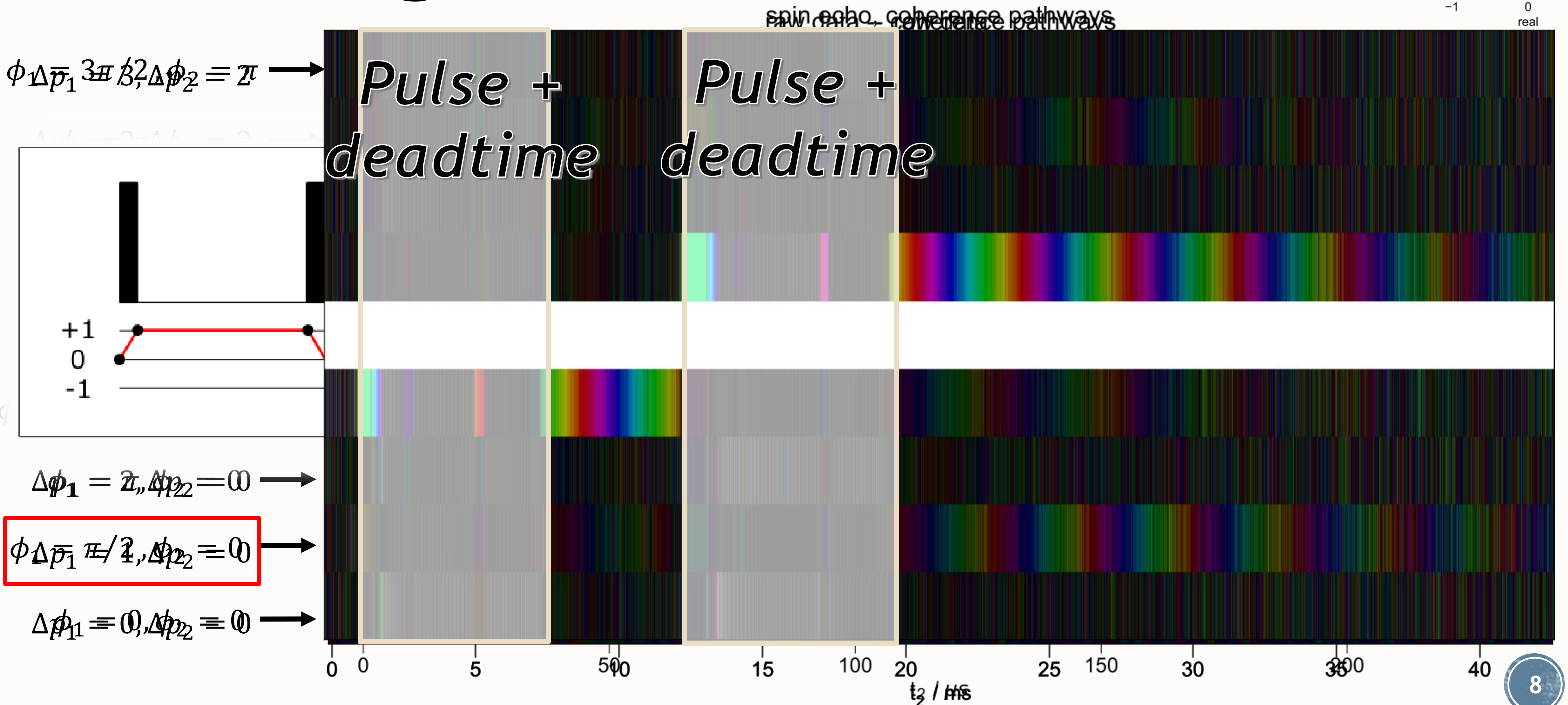
Add figures (and text) to a figure list:

- keep track of units of different datasets added to the figure
- assign names (usually matching titles) to the plots
- use the same code to generate pop-up windows, or drop it into a PDF (latex) lab notebook

%pylab inline

Plotting all the data

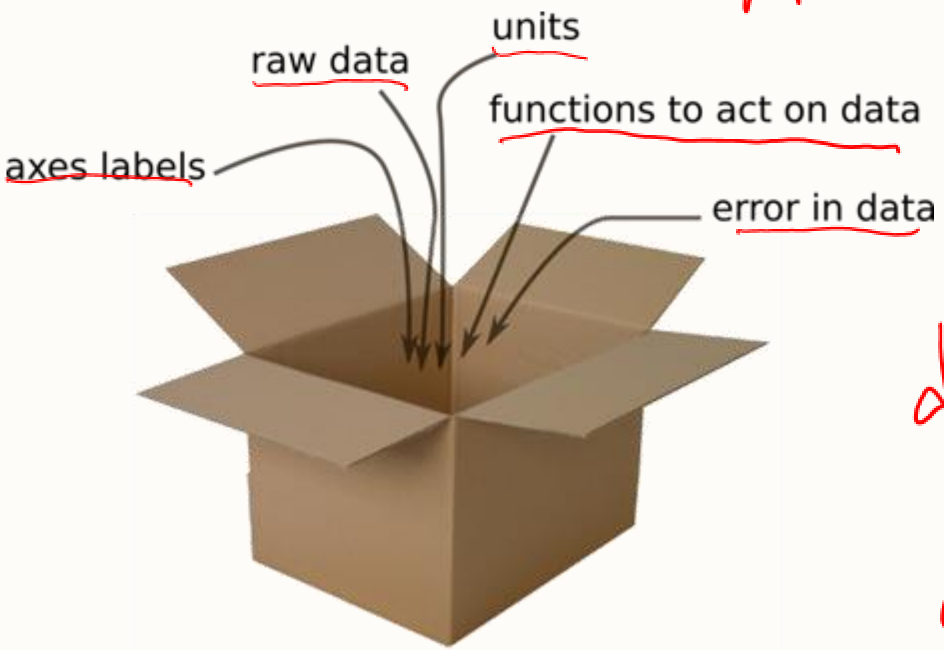




1. Bodenhausen, G. et al. *Journal of Magnetic Resonance (1969)* **1984**, *58* (3), 370–388.

nddata objects

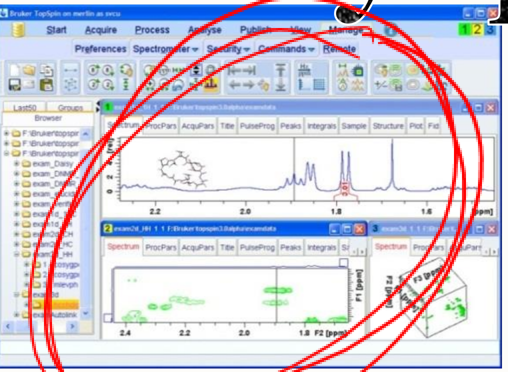
PySpecData nddata objects



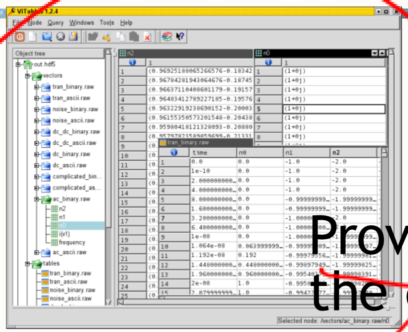
Store all the information we might want for a plot (and more) in one “container” (**object**) → call our container `nddata`.

```
d-shape = ndshape(d)
d_shape += [('T', 10)]
d_combined = d_shape.alloc()
d = nddata(~~~)
```

PySpecData nddata objects



proprietary data formats



database

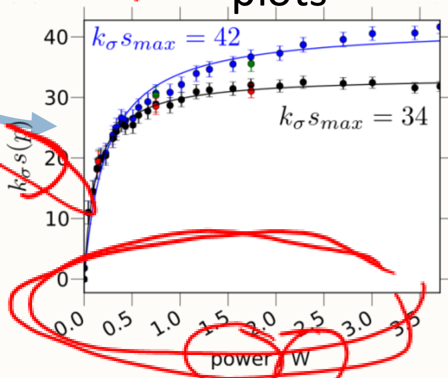
Provides an opportunity to structure the data.

“Nd-data”

d

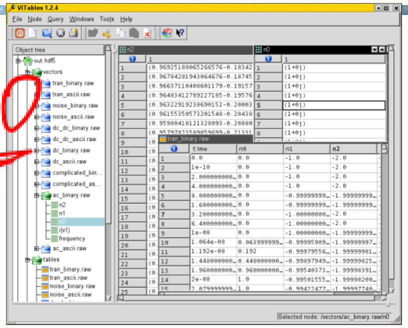
plot(d)

plots



PySpecData nddata objects

proprietary data formats



database

sc.gain=10

"Nd-data"

c = a+b

Also → facilitate manipulation and acquisition of data

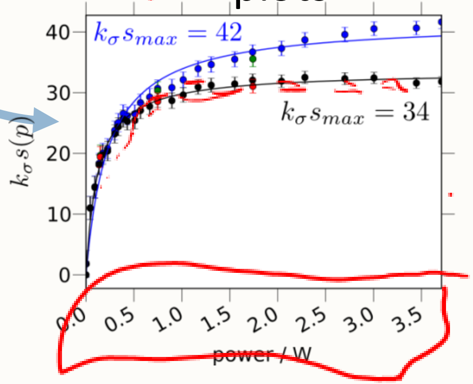
Technically:

- Methods
- Operator overloading
- Properties

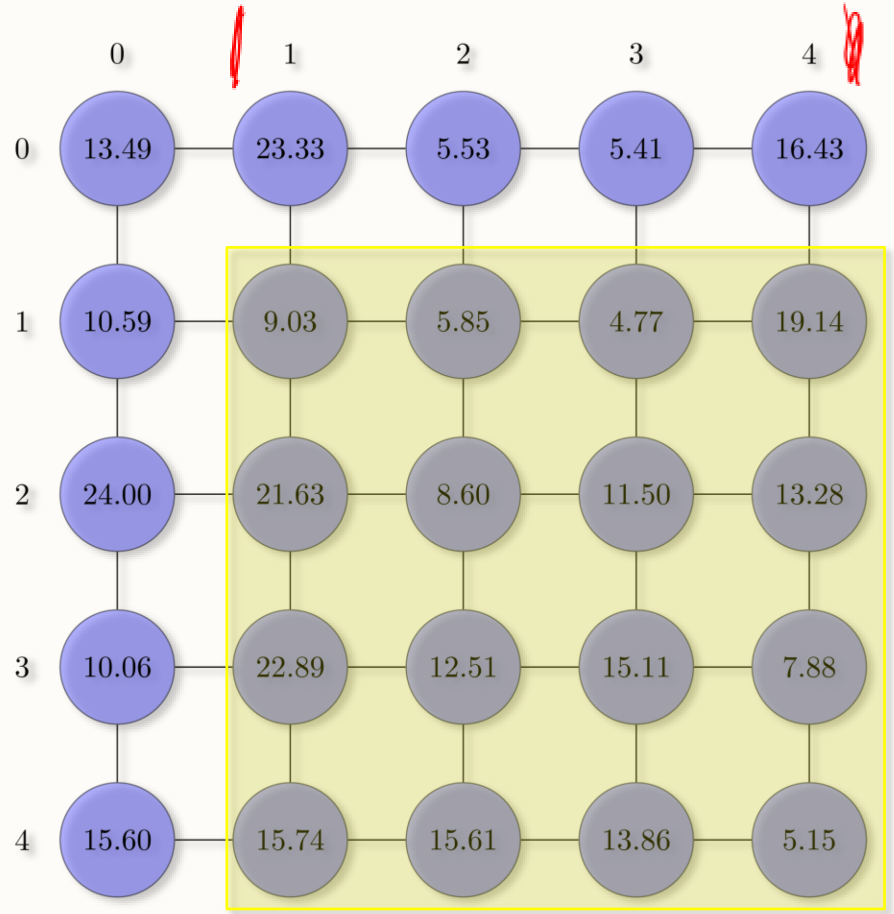
instruments

Fourier transform, error propagation, curve fit, etc.

plots



Standard numpy arrays (ndarray objects)



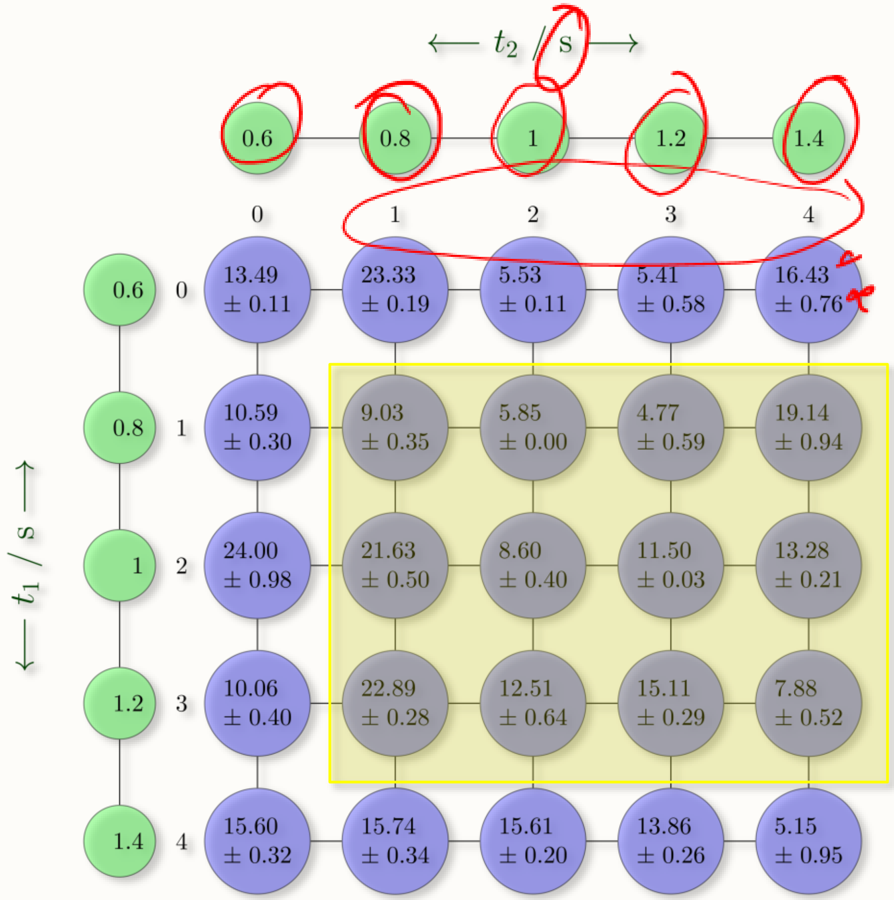
Say this is called `d`, then

```
d_sliced = d[1:,1:4]
```

slices out relevant data – *e.g.* for:

- frequency filtering
- selecting portions of time axis that actually contain data

PySpecData nddata objects



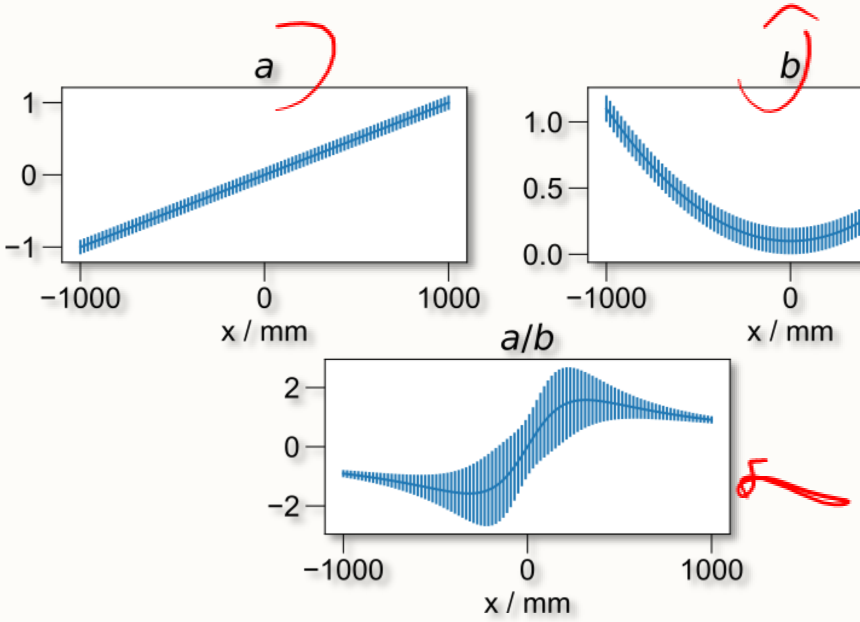
Named axes help avoid confusion:

```
d_sliced =
d['t1',1:4]['t2',1:]
```

We can slice by the *frequency* rather than the *index number*:

```
d_sliced =
d['t1':(0.8,1.2)]['t2':(0.8,
None)]
```

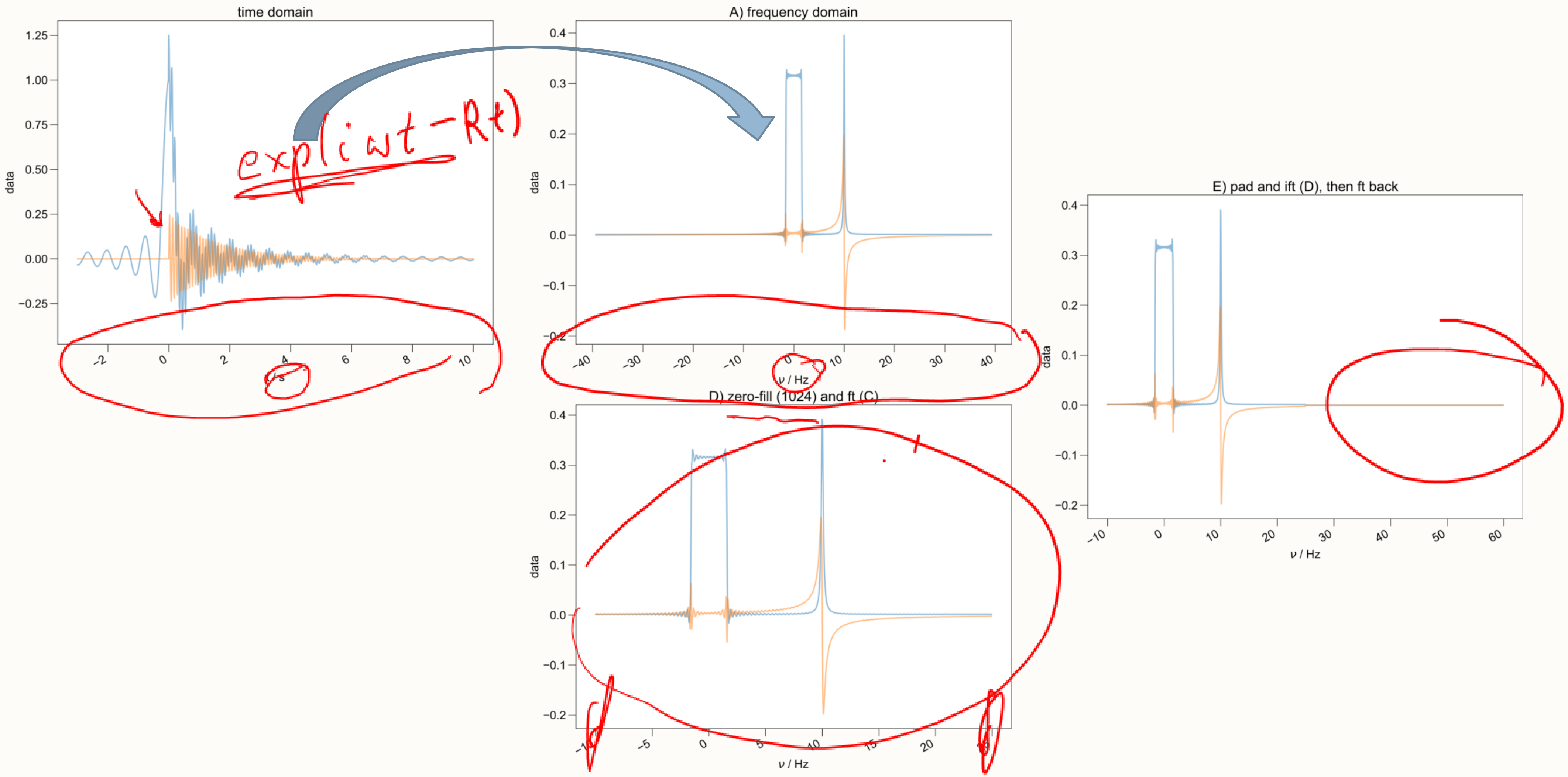
Automatic Propagation of Error



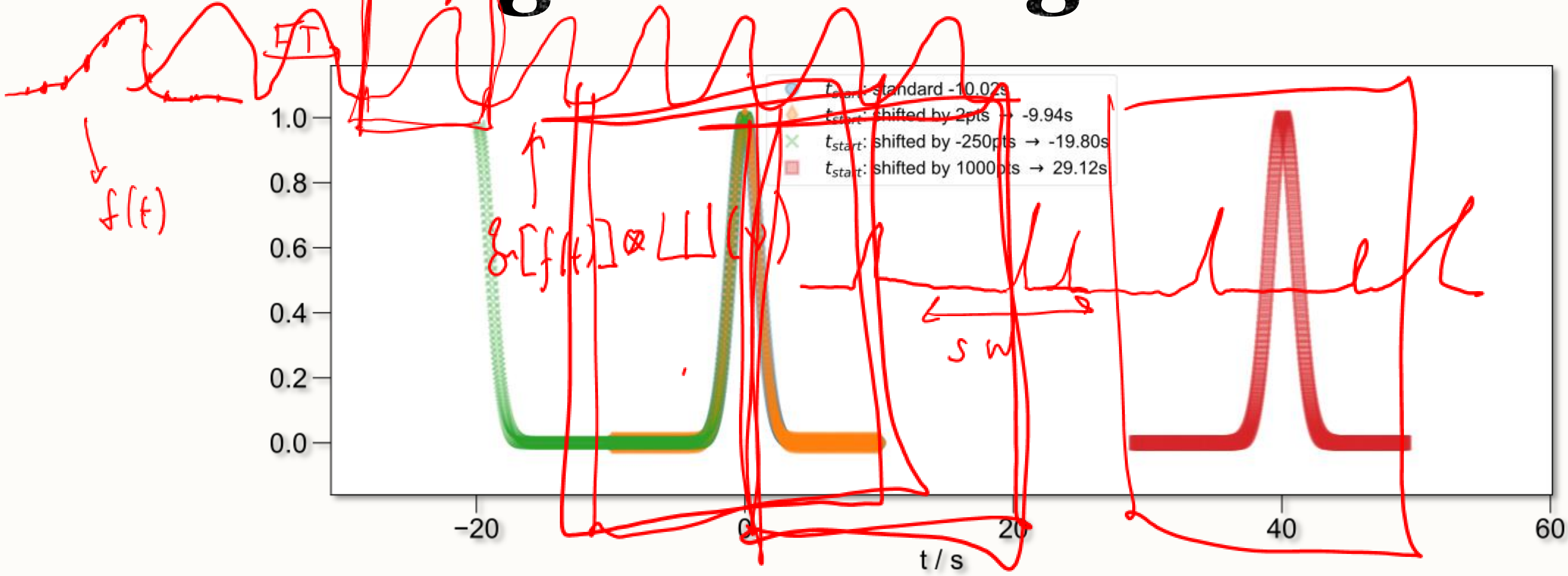

```
from pyspecdata import *
fl = figlist_var()
a = nddata(r_[-1:1:1j],'x').set_units('x','m')
a.set_error(0.1)
b = a.C
b.run(lambda t: t**2+0.1)
b.set_error(0.1)
print(b.get_error('x'))
fl.next('$a$',figsize=(4.5,2))
fl.plot(a)
fl.next('$b$',figsize=(4.5,2))
fl.plot(b)
fl.next('aob',figsize=(4.5,2))
fl.plot(a/b)
title('$a/b$')
fl.show()
```

Automatic Relabeling of axes

```
d.ft('t', shift=True)
```



Aliasing + Axis Registration



Powerful tools, such as looking at different "aliases":

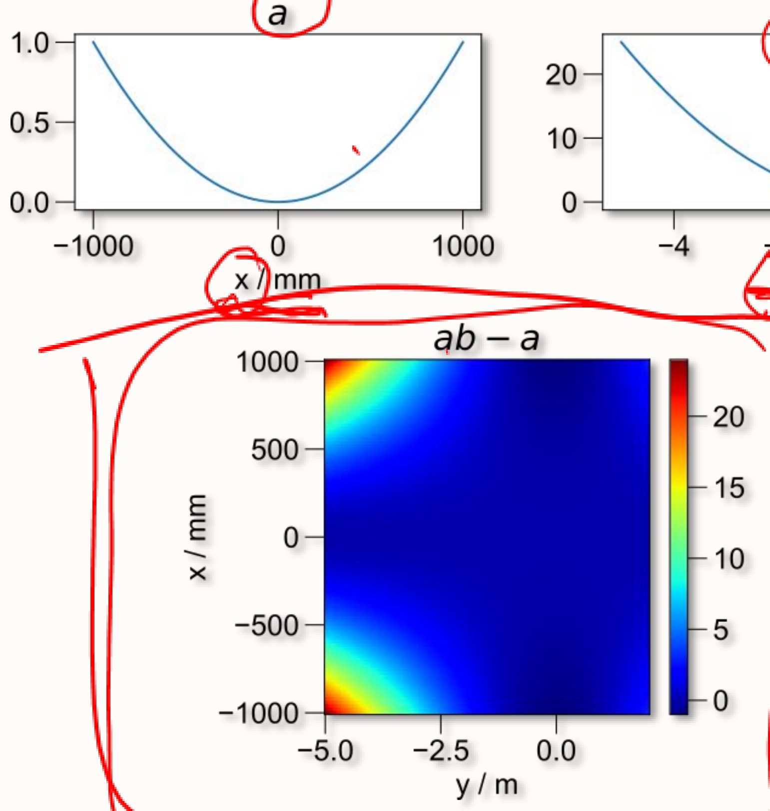
```
forplot.ft_clear_startpoints('t',
      t=new_startpoint, f='current')
```

More specifically, we can relabel our axes and interpolate onto a new axis:

```
s.setaxis('t2',lambda x: x-peak_center)
s.register_axis({'t2':0})
```

(first-order phase shift, but actually reflected in the data)

“Smart” Determination of Dimensionality




```
from pyspecdata import *
fl = figlist_var()
a = nddata(r_[-1:1:100j],'x').set_units('x','m')
a.run(lambda t: t**2)
b = nddata(r_[-5:2:200j],'y').set_units('y','m')
b.run(lambda t: t**2)
fl.next('$a$', figsize=(4.5,2))
fl.plot(a)
fl.next('$b$', figsize=(4.5,2))
fl.plot(b)
fl.next('$ab-a$', figsize=(4.5,4))
fl.image(a*b-a)
fl.show()
```

Methods

ph1 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
ph2 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

These are all *methods* → they appear to be *inside* the data, *e.g.*:

```
d = d.copy()
```

Manipulating attributes of the data:

- copy
- copy_props → copy the "other info"
- get_units → units of axis
- replicate_units
- set_units
- setaxis → manipulate the axis
  - can take a function as an argument

    ```
    s.setaxis('t2',lambda x: x-peak_center)
    ```
- set_prop → properties in a generic "other info" dictionary
- copyaxes
- get_error
- get_plot_color
- get_prop → opposite of "set_prop"

- hdf5_write
- like → "ones-like"
- labels → set multiple axis labels w/ a dictionary
- name → (of the data) used for HDF5 + plotting
- set_error

Reshaping:

- chunk → split dimension
- chunk_auto → split dimension based on record array
- item → single element to float or int
- rename → dimension label
- reorder
- smoosh → opposite of "chunk"
- squeeze → remove dimensions of size 1
- extend → zero padding

d.chunk('ph1',['ph2','ph1'],[4,4])

Methods

Utility Functions (benefited by naming):

- argmax, argmin → yields axis values (frequency/time)
- contiguous → takes a logical function, yields axis values (*e.g.* peak picking)
- dot
- fromaxis → generate an nddata from an axis
- integrate → uses x axis for numerical integration
- interp,invinterp
- polyfit
- *(overloaded)*: + - / exp and trig functions
  - error propagation
  - automatic reordering

General utilities:

- add_noise → for simulation
- cdf → cumulative dist. func.
- circshift
- cropped_log
- mean, mean_all_but, mean_nopop, mean_weighted
- run, run_nopop, runcopy → apply numpy functions in-place
- sum, sum_nopop, diff
- sort
- to_ppm

Methods

Plotting:

- contour
- histogram
- human_units
- matrices_3d
- mayavi_surf
- meshplot
- oldtimey
- unitify_axis
- plot_labels
- waterfall
- set_plot_color

Fourier:

- convolve
- fourier_shear
- ft
- ft_clear_startpoints
- ftshift
- get_ft_prop
- ift
- register_axis
- set_ft_prop

Methods

More internal/development:

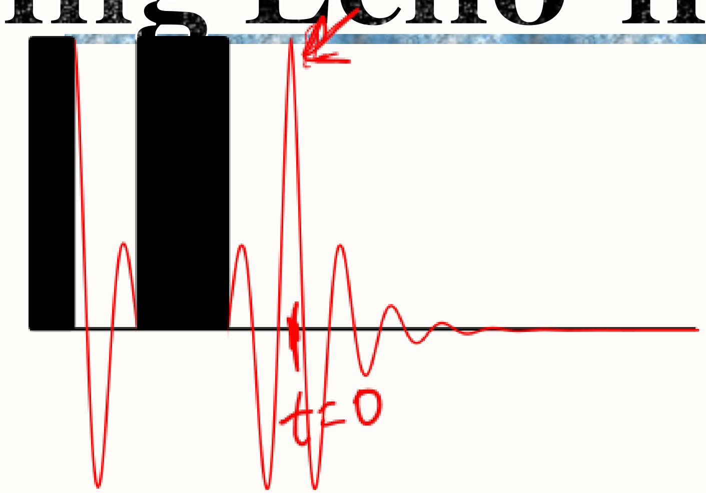
- aligndata → match shapes
- axn → (internal) number of named axis
- fld → generate dict of info about axes
- getaxis → ndarray for axis of a given dimension
- mkd ~nddata.popdim ~nddata.set_to

More complicated manipulations:

- extend_for_shear
- inhomog_coords
- linear_shear
- nnls → ILT
- secsy_transform
- secsy_transform_manual
- shear

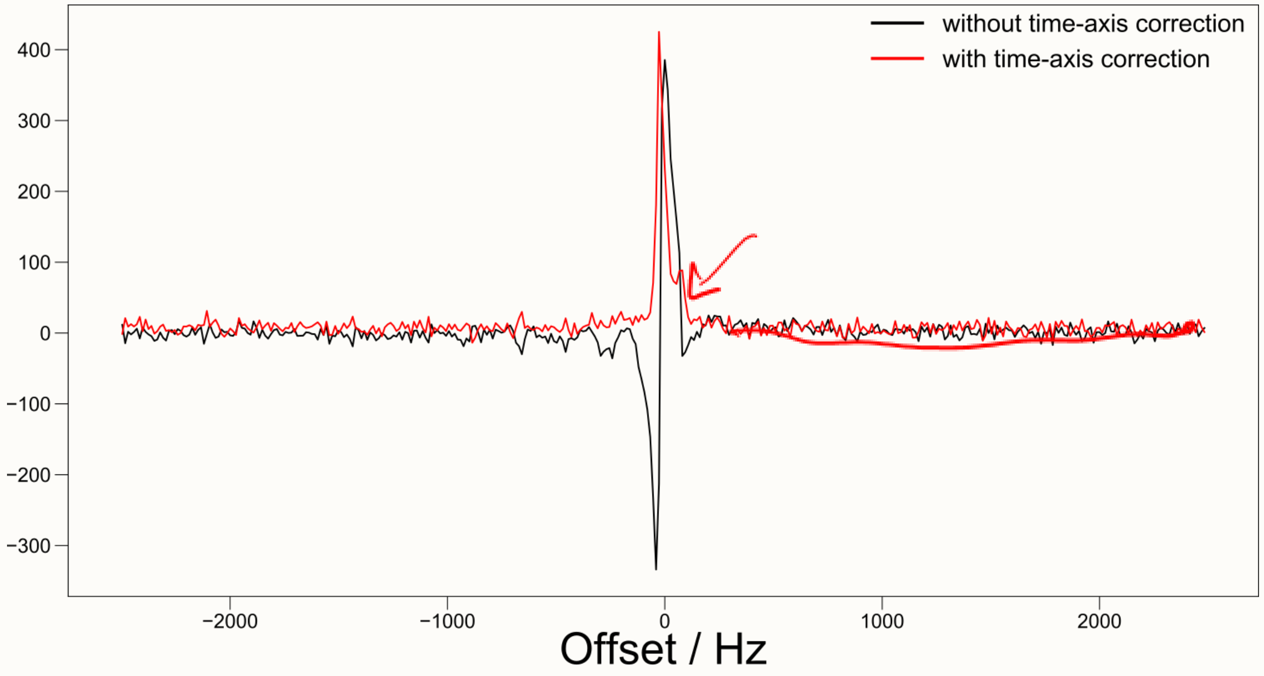
Some examples

Phasing Echo-like data



accomplished with:

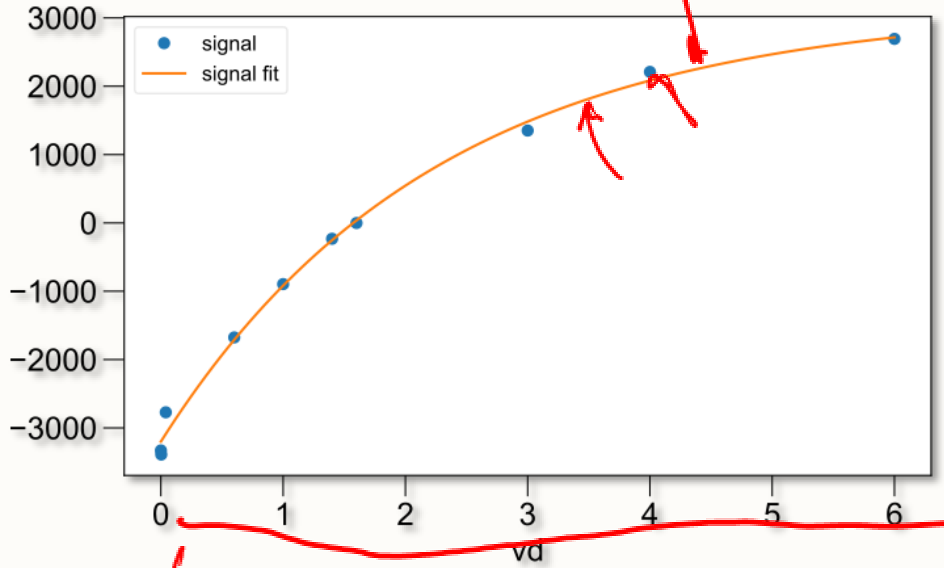
```
s.setaxis('t2',lambda x: x-
        peak_center)
s.register_axis({'t2':0})
```



Fitting T_1 data

def fitfunc(p,x):
return p[0]+(p[1]-p[2]))

f.eval(300)



$f(vd) = 3.15 \times 10^3 + (-3.20 \times 10^3 - 3.15 \times 10^3)e^{-\frac{vd}{2.25}}$

class `fitdata` inherits from `nddata`

this is python for:

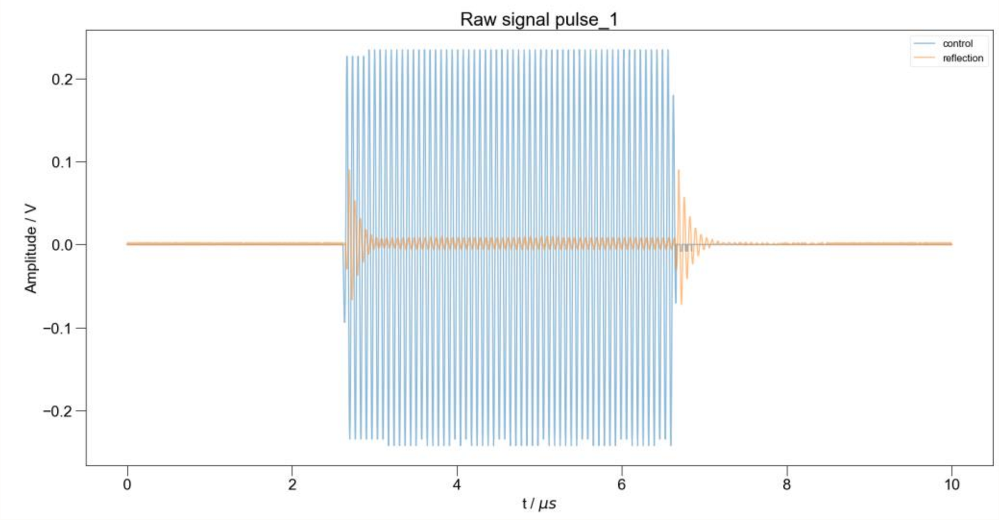
$$f(vd) = M_{inf} + (M_0 - M_{inf})e^{-\frac{vd}{T_1}}$$

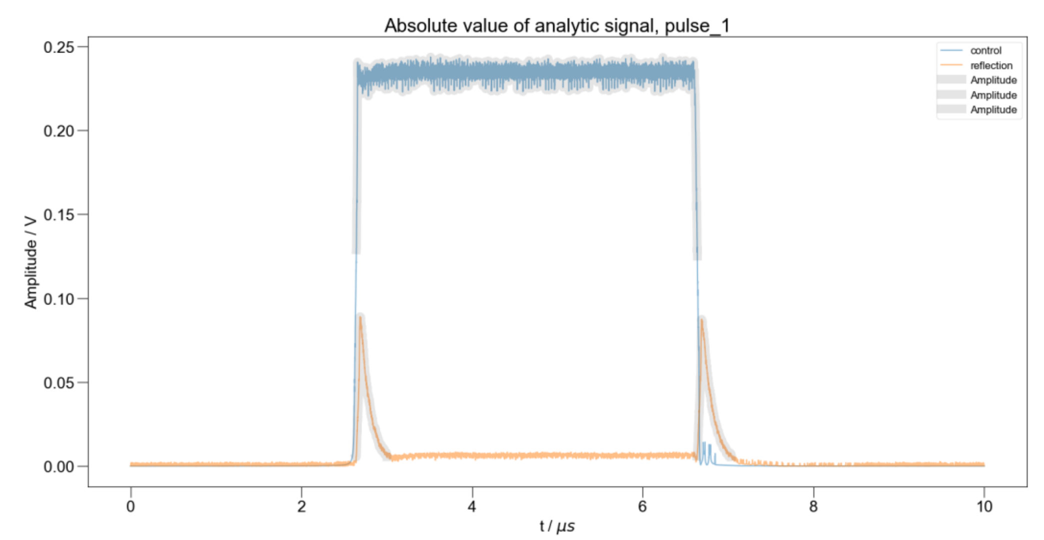
```
f = fitdata(s_sliced)
M0,Mi,T1,vd = sympy.symbols(
    "M_0 M_inf T_1 vd",
    real=True)
f.functional_form = Mi + (M0
    -Mi)*sympy.exp(-vd/T1)
```

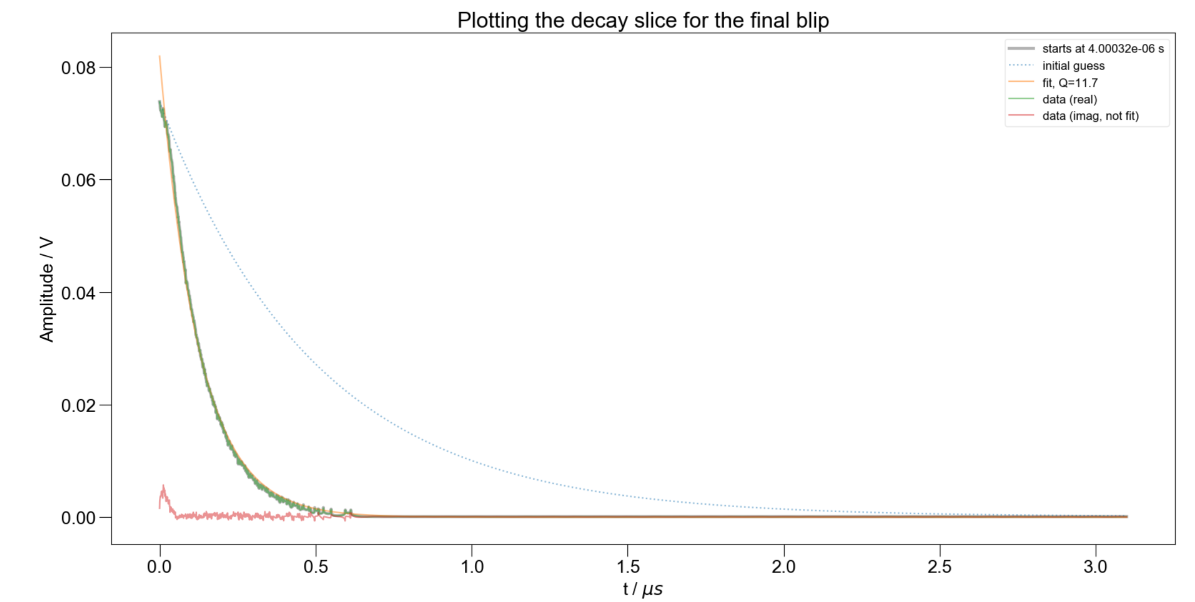
f.fit()

$$\boxed{Ae^{-R_2 t}}$$

Determining the transfer function from the pulse reflection



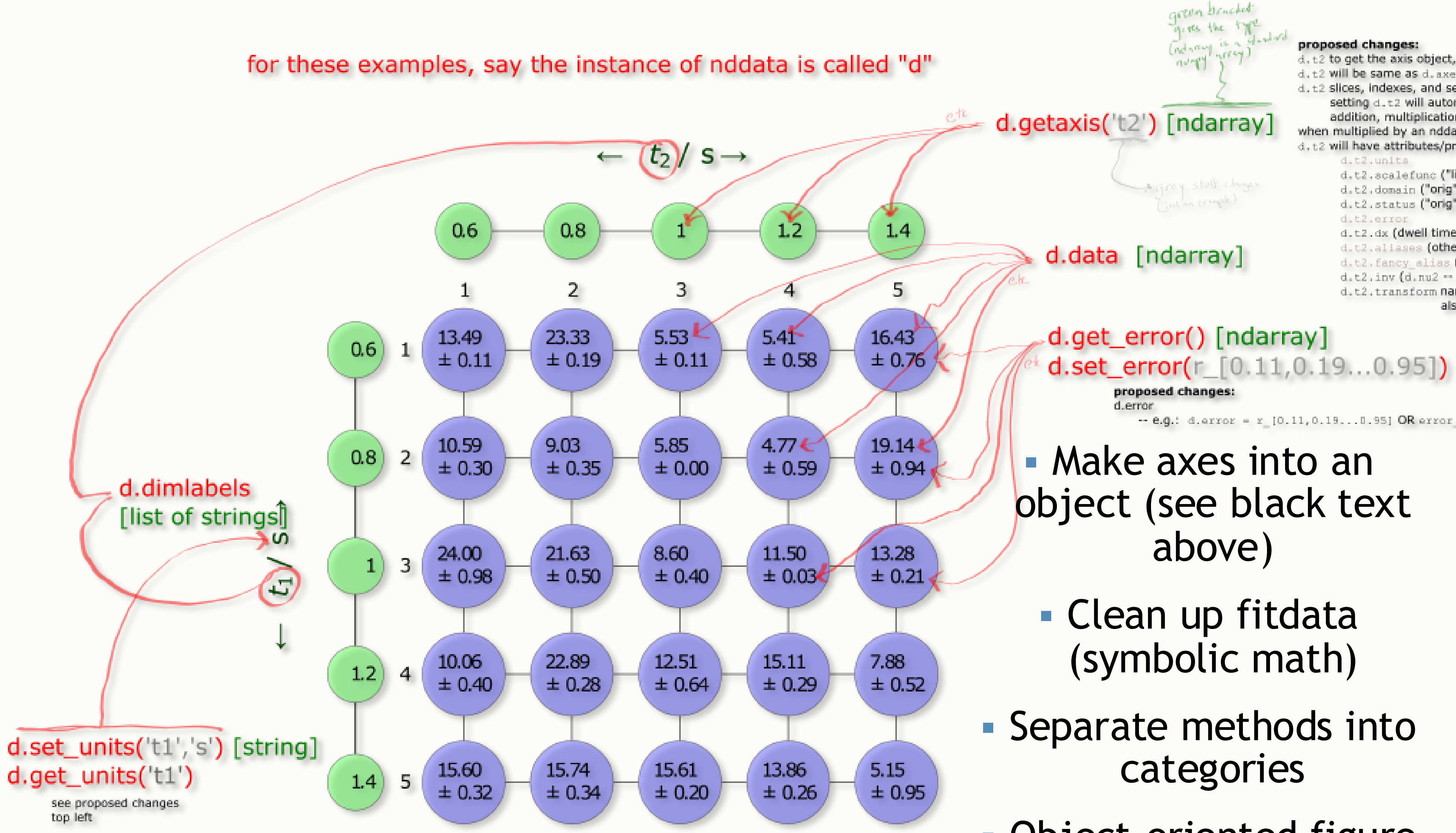




Future Directions

proposed changes:

`d.t2` to get the axis object, where `"t2" in d.dimlabels`
`d.t2` will be same as `d.axes["t2"]`
`d.t2` slices, indexes, and setslice like a normal numpy array
    setting `d.t2` will automatically determine `d.t2.scalefunc` and `dx`
    addition, multiplication, division, log/exp/sin are overloaded
when multiplied by an nddata, `d.t2` will behave like an nddata (`d.fromaxis('t2')`)
`d.t2` will have attributes/properties -- only items in brown would be modifiable:
- `d.t2.units`
- `d.t2.scalefunc` ("linear" would be a scalar, for log spacing, the function `log10`, or else `None`)
- `d.t2.domain` ("orig" vs. "trans" -- is t2 the original or transformed domain)
- `d.t2.status` ("orig" vs. "trans" -- current status of the data)
- `d.t2.error`
- `d.t2.dx` (dwell time or frequency spacing or `None` if nonlinear)
- `d.t2.aliases` (other names: `["t_2","t"]` -- here `d.t` would be same as `d.t2`)
- `d.t2.fancy_alias` (for plotting)
- `d.t2.inv` (`d.nu2` -- simple name of the conjugate axis)
- `d.t2.transform` name of method used to transform ("ft", "ift", "nnls", "nus", etc.)
  also `d.t2.do_transform()`



proposed changes:
d.error
-- e.g.: `d.error = r_[0.11,0.19...0.95]` OR `error_array = d.error`

- Make axes into an object (see black text above)
- Clean up fitdata (symbolic math)
- Separate methods into categories
- Object-oriented figure list

Figure list upgrades

Take advantage of “figure lists” while avoiding some of the disadvantages:

Build the new system around a list of *visualization objects*.

In the simplest example, a visualization object corresponds to e.g. a single matplotlib axis; but it can also correspond to multiple axes, as when we want to plot multi-dimensional data, or when we want to show the projections of 2D data.

By breaking the plotting process into several methods, we can automate the process, while also keeping it maximally flexible.

From the end-user’s perspective, plotting is achieved by

1. optionally modifying an existing plotting object class to get different properties than the default: the default classes available are image, plot, contour, and surface
2. creating instances of the plotting classes inside inside the figure list, in order of appearance (note that sometimes, it will make sense to first initialize blank plots, and drop data into them if the order of plotting and data generation are different)
3. adding nddata objects (or numpy objects, which are converted to nddata objects) to the plotting instances (quite literally)

```
with figlist_var('filename.pdf') as fl:
    pl_first = fl.next()
    pl_first += d_rough.real
```

4. if interactive plotting is employed, the nddata themselves are updated, and then the figurelist update method is called.