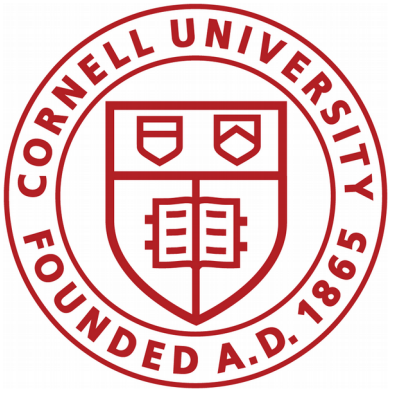


CBPM3 display

Antoine, Jim, Suntao

CBPM meeting

March 12, 2021

New approach

Instead of minimizing a global merit function of (x, y):

$$\sqrt{\Sigma_i (1 - \frac{f_i(x,y)}{b_i})^2}$$

→ solve simultaneously a set of 4 equations (root finding):

$$\begin{aligned}
1 - \frac{f_1(x,y)}{b_1}) &= 0, \\
1 - \frac{f_2(x,y)}{b_2}) &= 0, \\
1 - \frac{f_3(x,y)}{b_3}) &= 0, \\
1 - \frac{f_4(x,y)}{b_4}) &= 0
\end{aligned}$$

Levenberg-Marquardt solver

Among the many solvers, we use the default *Levenberg-Marquardt* which is the one also used by the CESRV code

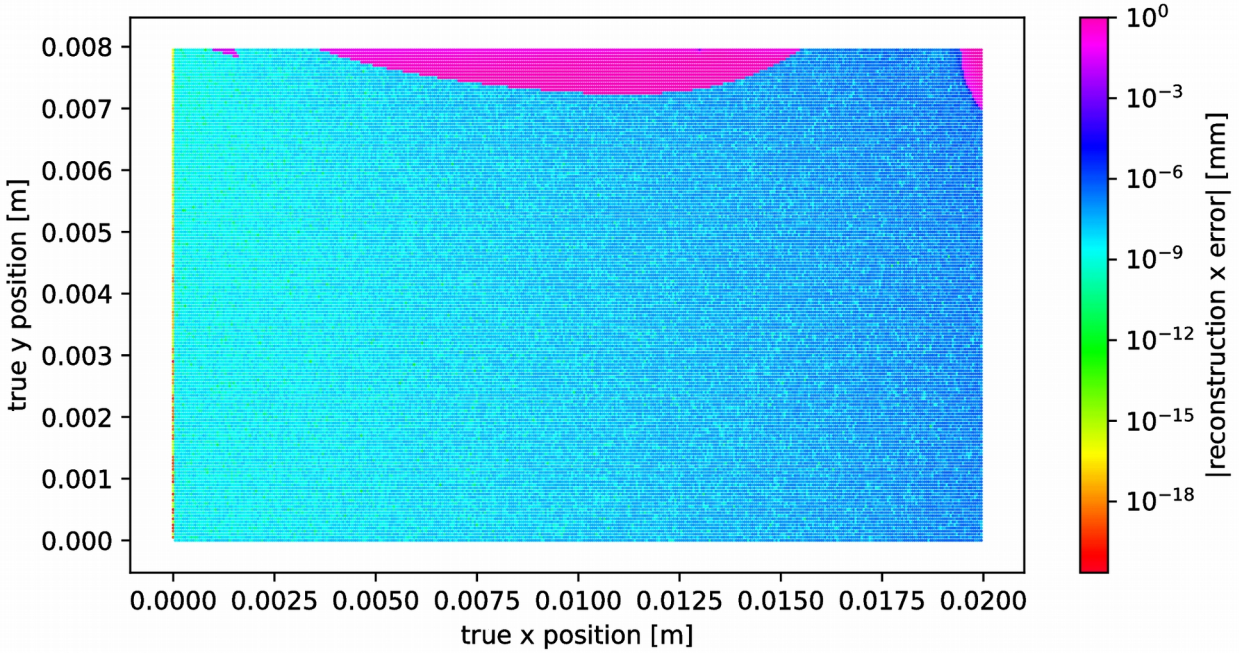
```
def fun(x, *args):
    f = [
    (1-f_norm_b1(x[0], x[1])[0]/args[0]) , #  button 1
    (1-f_norm_b1(-1*x[0], x[1])[0]/args[1]),  # button 2
    (1-f_norm_b1(x[0], -1*x[1])[0]/args[2]),  # button 3
    (1-f_norm_b1(-1*x[0], -1*x[1])[0]/args[3])  # button 4
    ]
    return f

result = optimize.root(
        fun,
        initial_guess,
        args=(round(button[0], 9), round(button[1], 9), round(button[2], 9), round(button[3], 9)),
        method='lm',
        jac=False)
```

Initial guess = (x, y) computed using k_x and k_y

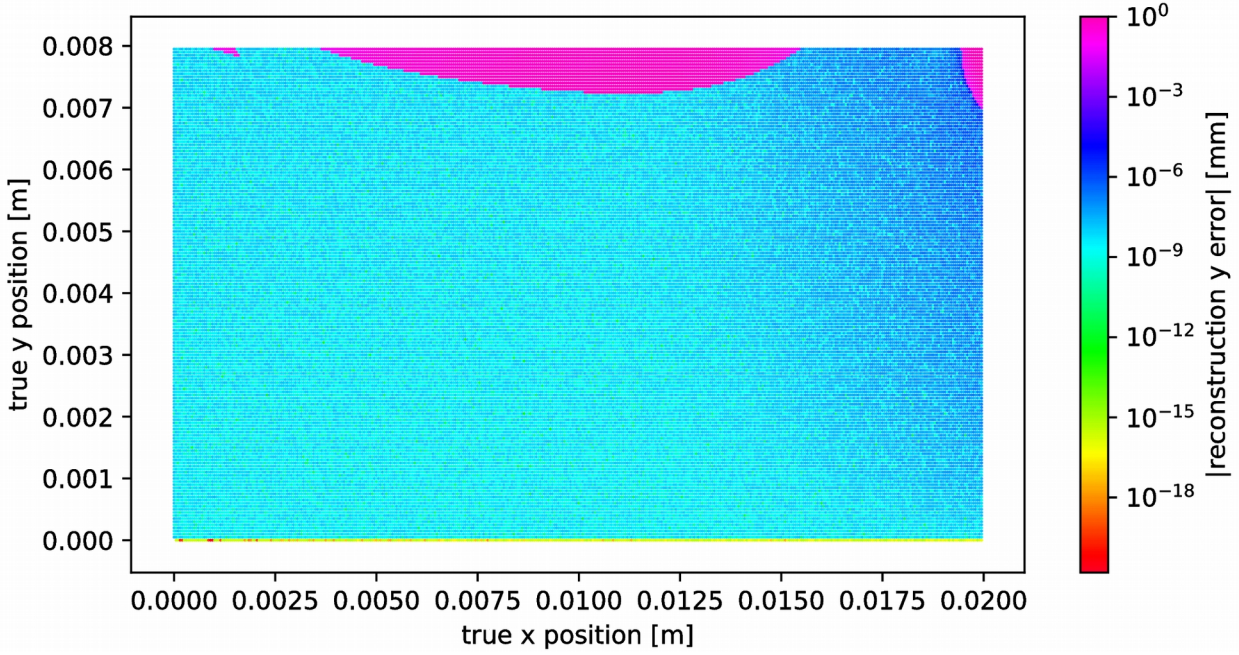
Out-of-the box results

color bar clipped [-inf, 1 mm]

bpm_chessu_xyp.txt



average solving time = 1.6 ms

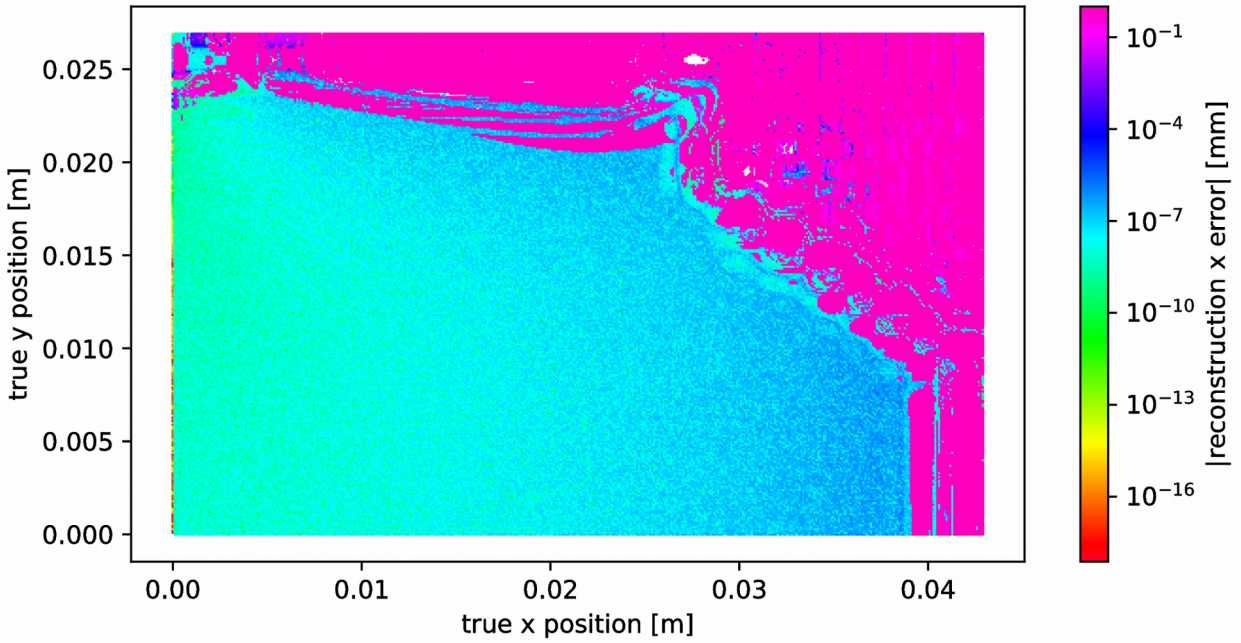


100 μm step size

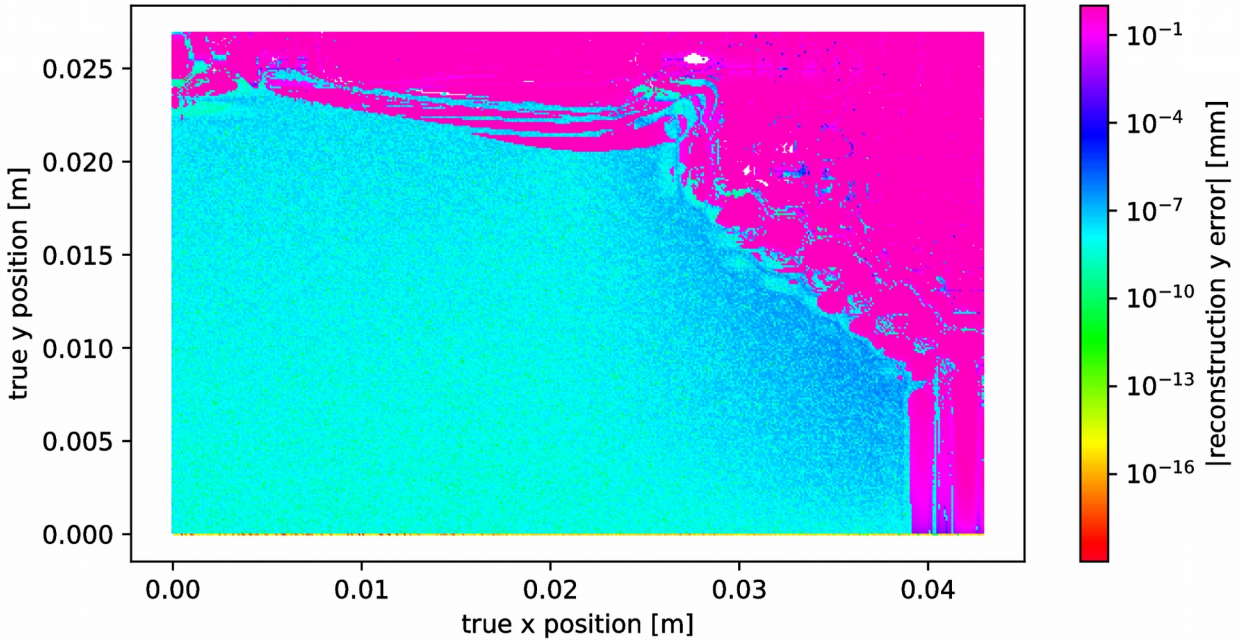
Out-of-the box results

color bar clipped [-inf, 1 mm]

bpm_arc_xyp.txt



average solving time = 2.2 ms



100 μm step size

Toward more realism

Toward more realism

So far, it has been an ideal world with “infinite” precision:

- ✗ generate “measured” button amplitudes from the **normalized** (quintic) extrapolated look-up table
- ✗ reconstruct (x, y) using the normalized (quintic) extrapolated look-up table

What if instead we:

- ✗ generate “measured” button amplitudes from the **non-normalized** (quintic) extrapolated look-up table
- ✗ reconstruct (x, y) using the normalized (quintic) extrapolated look-up table

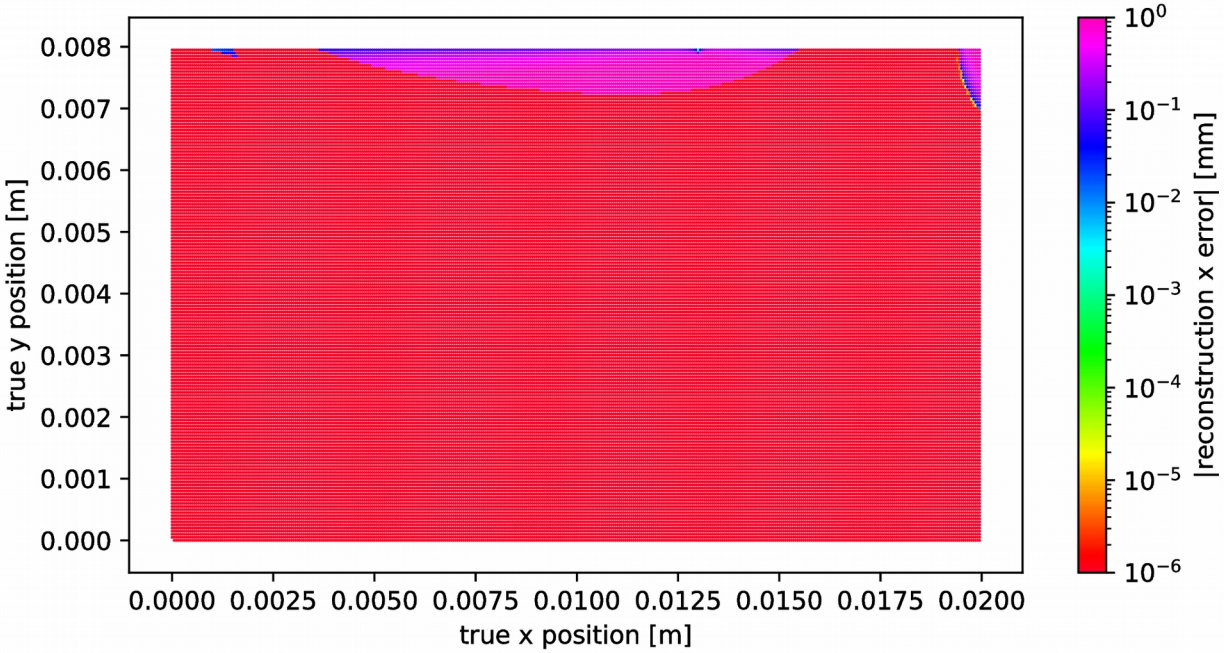
→ limitation due to need to normalize input data?

Normalized vs un-normalized input

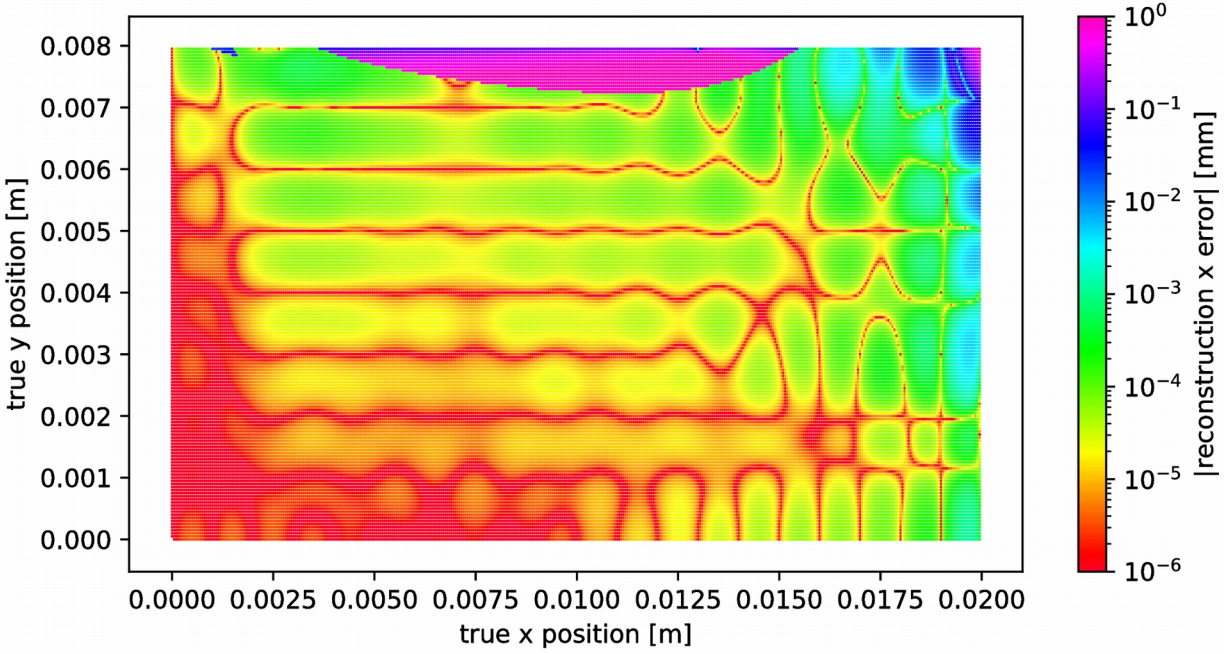
color bar clipped [1 nm, 1 mm]

bpm_chessu_xyp.txt

no need to normalize input data



need to normalize input data



50 μm step size

Toward more realism

So far, it has been an ideal world with “darn good” precision:

✗ generate “measured” button amplitudes from the non-normalized (quintic) extrapolated look-up table

✗ reconstruct (x, y) using the normalized (quintic) extrapolated look-up table

What if instead we:

✗ generate “measured” button amplitudes from the non-normalized (**quintic**) extrapolated look-up table

✗ reconstruct (x, y) using the normalized (**cubic**) extrapolated look-up table

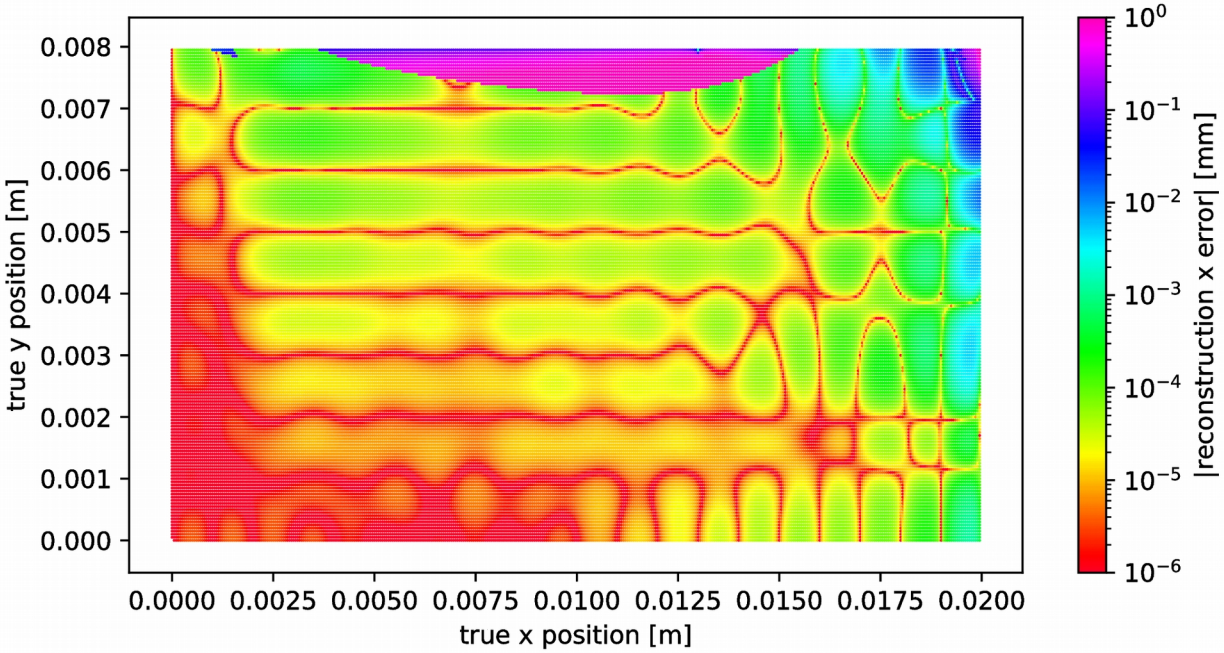
→ limitation due to need to normalize input data and look-up table?

Normalization + look-up table

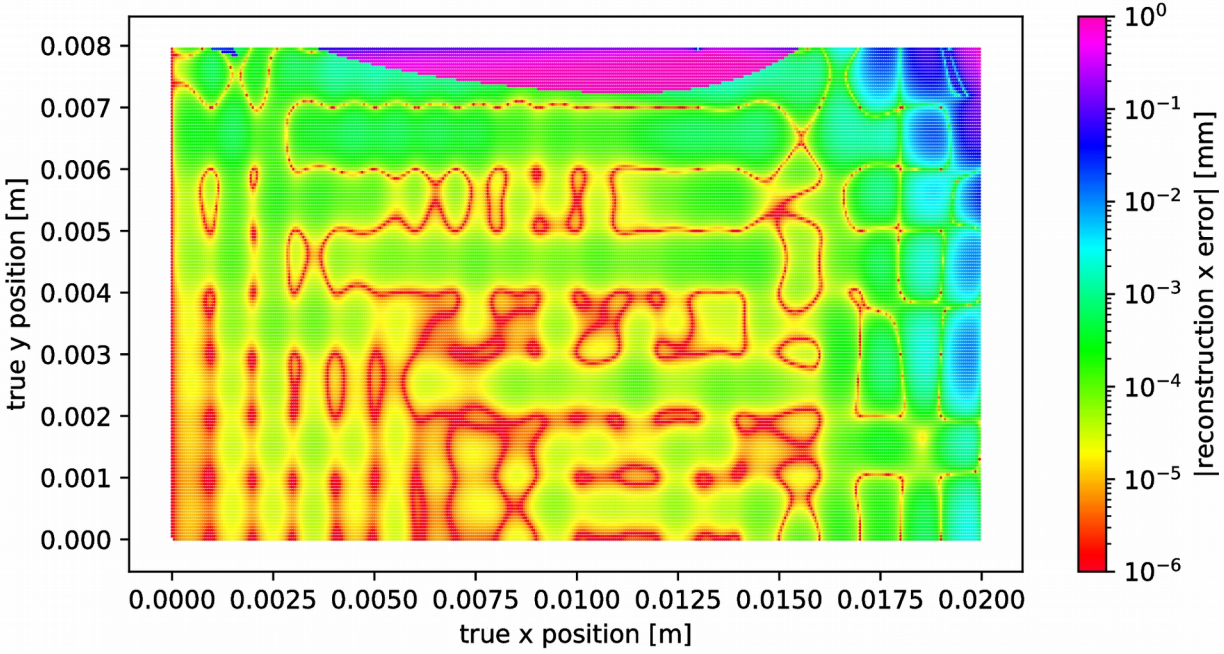
color bar clipped [1 nm, 1 mm]

bpm_chessu_xyp.txt

normalization + quintic look-up table



normalization + cubic look-up table



50 μm step size

Toward more realism

So far, it has been an ideal world with “darn good” precision:

✗ generate “measured” button amplitudes from the non-normalized (quintic) extrapolated look-up table

✗ reconstruct (x, y) using the normalized (cubic) extrapolated look-up table

What if instead we:

✗ generate “measured” button amplitudes from the non-normalized (**quintic**) extrapolated look-up table with **integer amplitude ranging from 20,000 for (x,y)=(0,0) to about 80,000 for (x,y) at button location** (in reality amplitude clips at 32,000)

✗ reconstruct (x, y) using the normalized (cubic) extrapolated look-up table

→ limitation due to need to normalize input data, look-up table and amplitude precision?

Toward more realism

So far, it has been an ideal world with “darn good” precision:

✗ generate “measured” button amplitudes from the non-normalized (quintic) extrapolated look-up table

✗ reconstruct (x, y) using the normalized (cubic) extrapolated look-up table

What if instead we:

We’ll use this configuration for all the remainder studies in today’s presentation

✗ generate “measured” button amplitudes from the non-normalized (**quintic**) extrapolated look-up table with i**nteger amplitude ranging from 20,000 for (x,y)=(0,0) to about 80,000 for (x,y) at button location** (in reality amplitude clips at 32,000)

✗ reconstruct (x, y) using the normalized (cubic) extrapolated look-up table

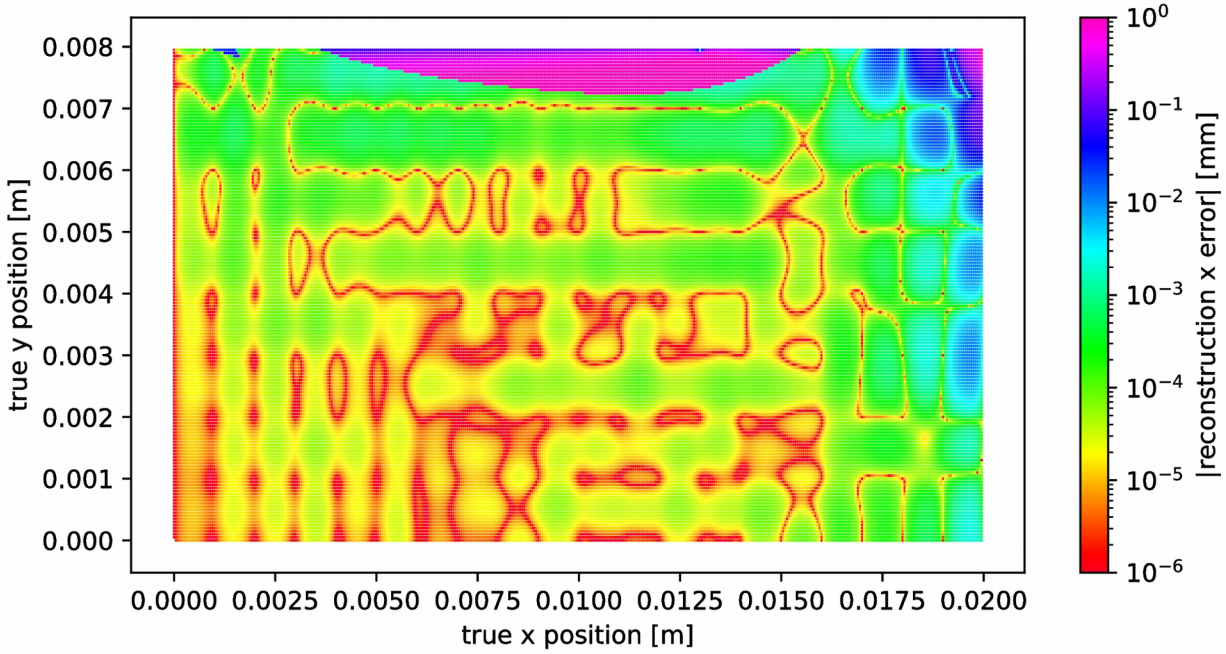
→ limitation due to need to normalize input data, look-up table and amplitude precision?

Normalization + look-up table + amplitude precision

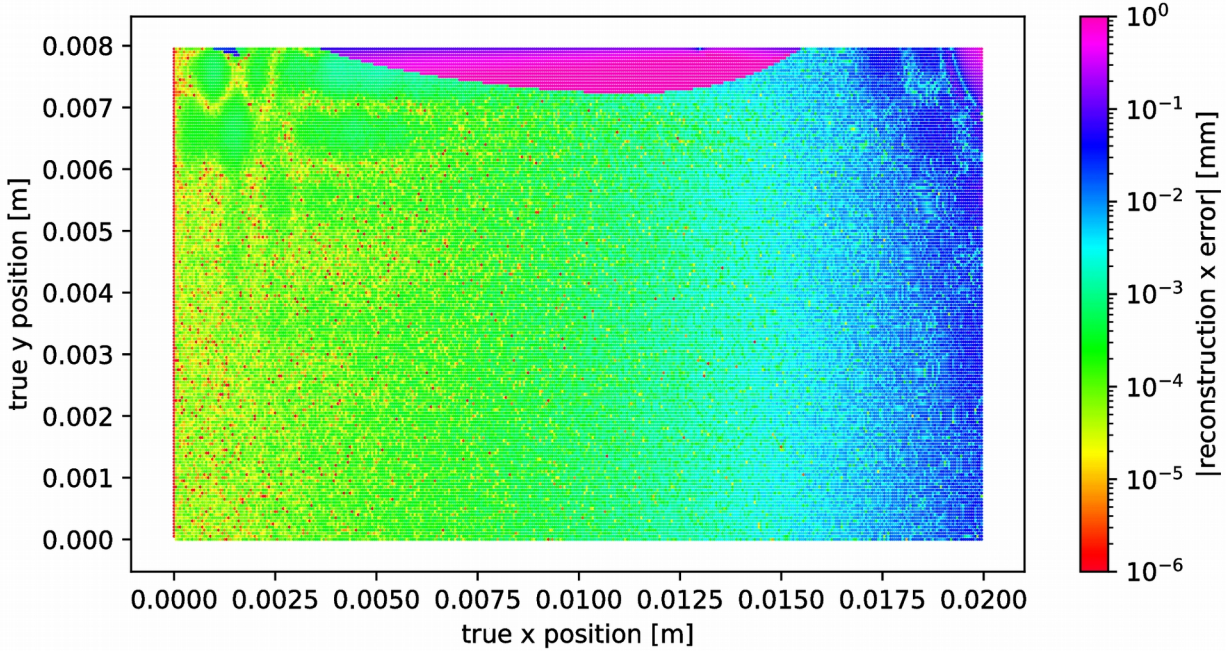
color bar clipped [1 nm, 1 mm]

bpm_chessu_xyp.txt

normalization + cubic look-up table



normalization + cubic look-up table + limited amplitude precision



50 μm step size

CESRV vs Python

CESRV vs Python

Same simulated button data is fed to both CESRV and Python code and (x,y) reconstruction performance is compared

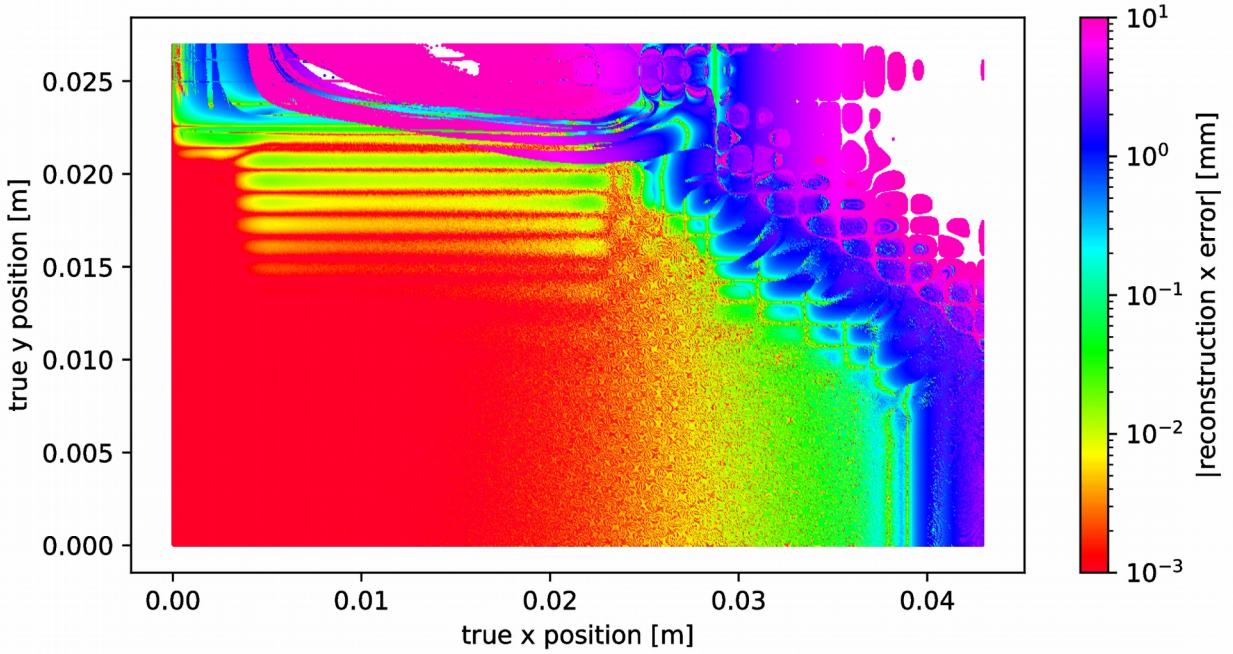
CESRV vs Python

color bar clipped [1 um, 10 mm]

bpm_arc_xyp.txt

Python

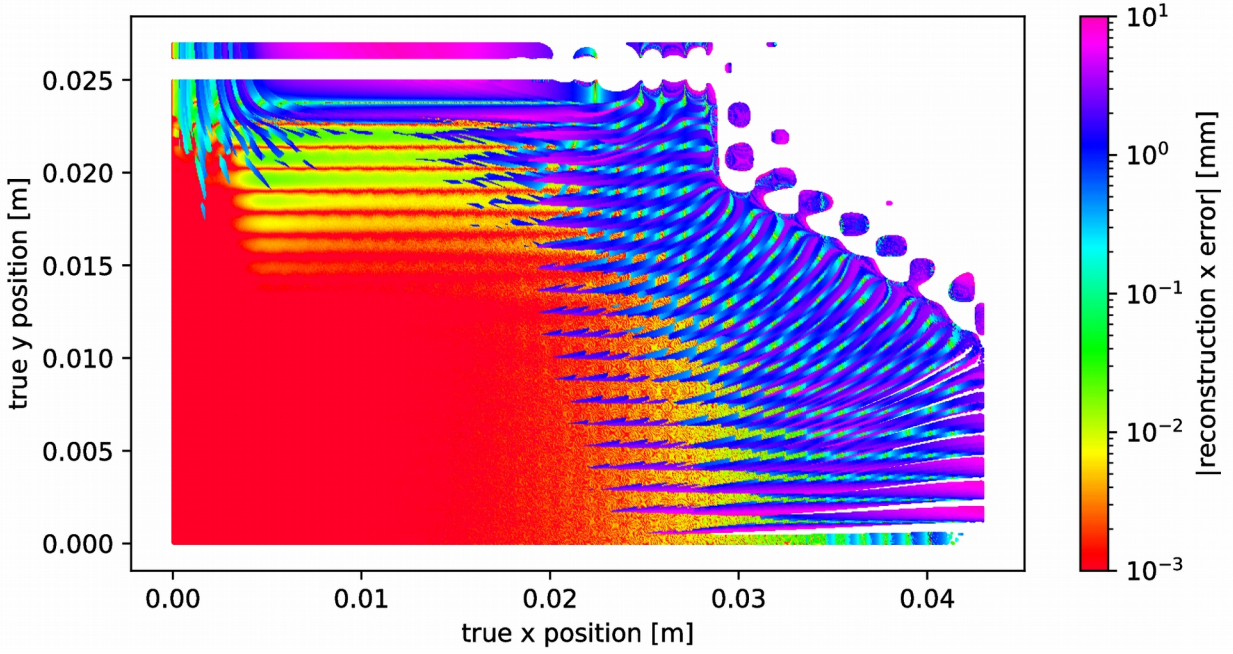
(white dots are when minimization fails)



average solving time = 2.5 ms

CESRV

(white dots are when minimization fails)



average solving time = 0.04 ms

50 μm step size

Some notes about CESRV non-lin as far as I understand

By default it seems the minimization chi^2 is not one of the available output (it is a Fortran `optional` in the subroutine `nonlin_bpm_minimize`)

When minimization fails → quad offset values are returned. Only notice of failure is a print-out (no chi^2 check is performed before returning final value to user):

```
109   param2 = param
110   call nonlin_mrqmin(but,sig,param,maska,covar,alpha,chisq,mrq_func,alamda)
111   if (nonlin_mrqmin_error) then
112     print *, "Bad data at BPM", i_bpm
113     exit
114   end if
115 end do
```

Given the above: checking minimization convergence was done in a “hacky” way using the fact that the code returns the quad offset value when the minimization fails

Would be valuable to pipe out the chi^2 from `nonlin_bpm_minimize` to `nonlin_orbit` so it is available alongside (x, y)

```
117 if (.not. nonlin_mrqmin_error) then
118    x = param
119    if(present(chi2_out)) chi2_out = chisq
120 else
121    x = 0.
122    if(present(chi2_out)) chi2_out = -1
123 end if
```

Next steps

Use (x,y) ↔ amplitude generated directly from Poisson and feed that to both Python and CESRV to test further the methods

Speed-up Python code

Use realistic instrumental errors

A practical use of the Python code

A practical use

During the previous CBPM3 meeting, the following idea was brought up:

Can we use 3 buttons instead of 4 to reconstruct the beam position?

For instance, X2D has one (long standing) problematic button, and this button is used by CESRV for the position reconstruction. Instead of a system of four equations, can we use:

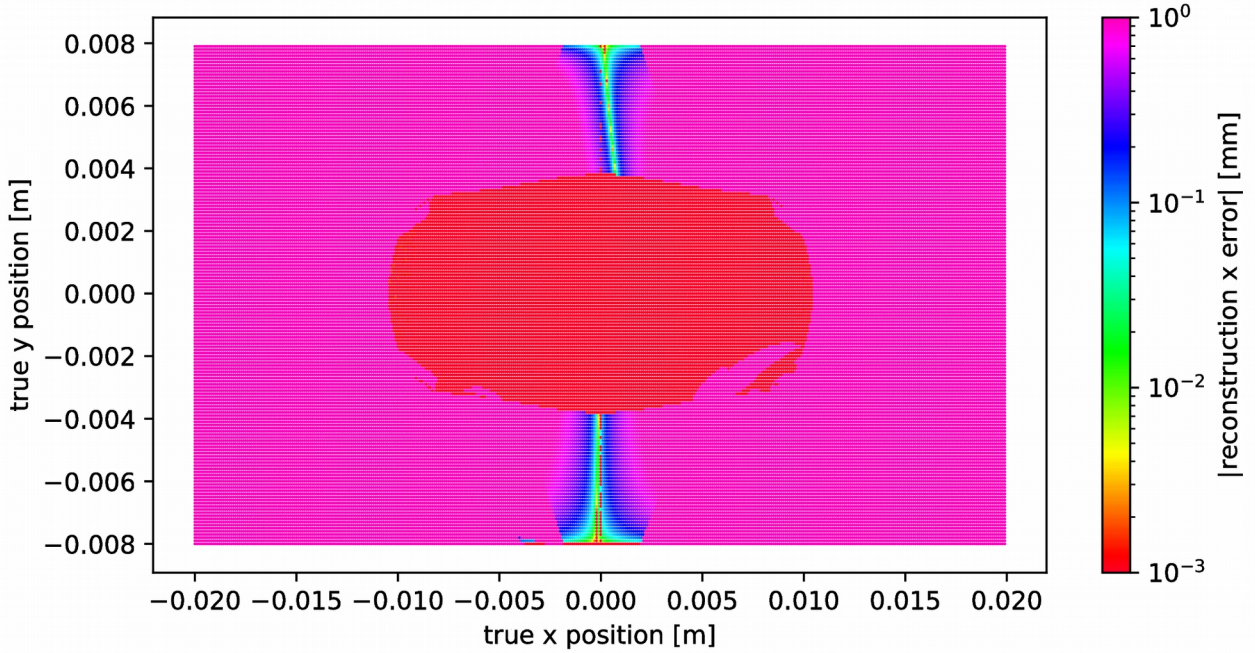
$$1 - \frac{f_1(x,y)}{b_1}) = 0,$$
$$1 - \frac{f_2(x,y)}{b_2}) = 0,$$
$$1 - \frac{f_3(x,y)}{b_3}) = 0$$

? Probably yes since the system is still well enough constrained

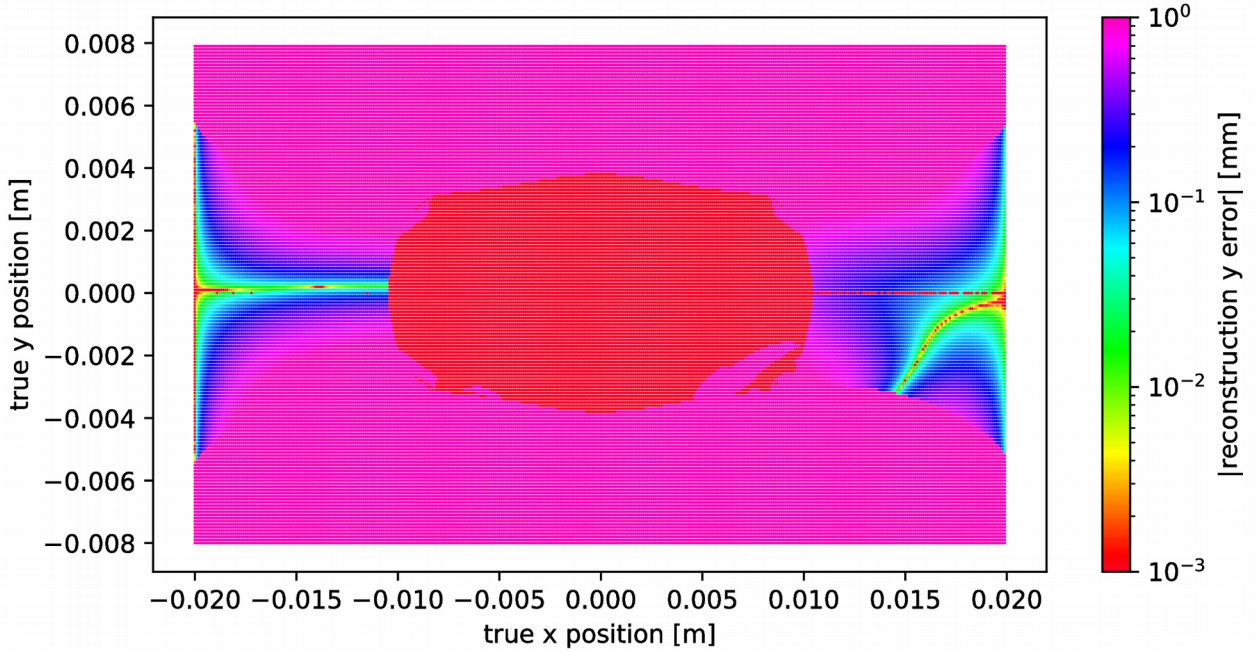
Dropping button 4

color bar clipped [1 um, 1 mm]

bpm_chessu_xyp.txt



average solving time = 1.0 ms

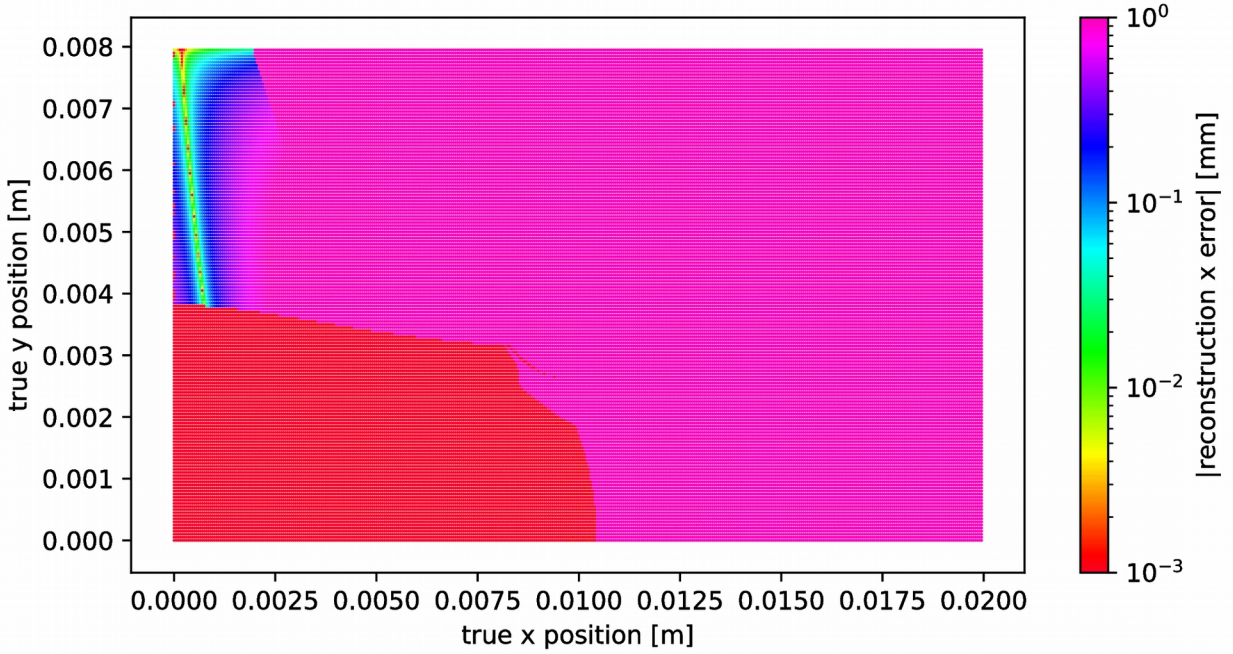


100 µm step size

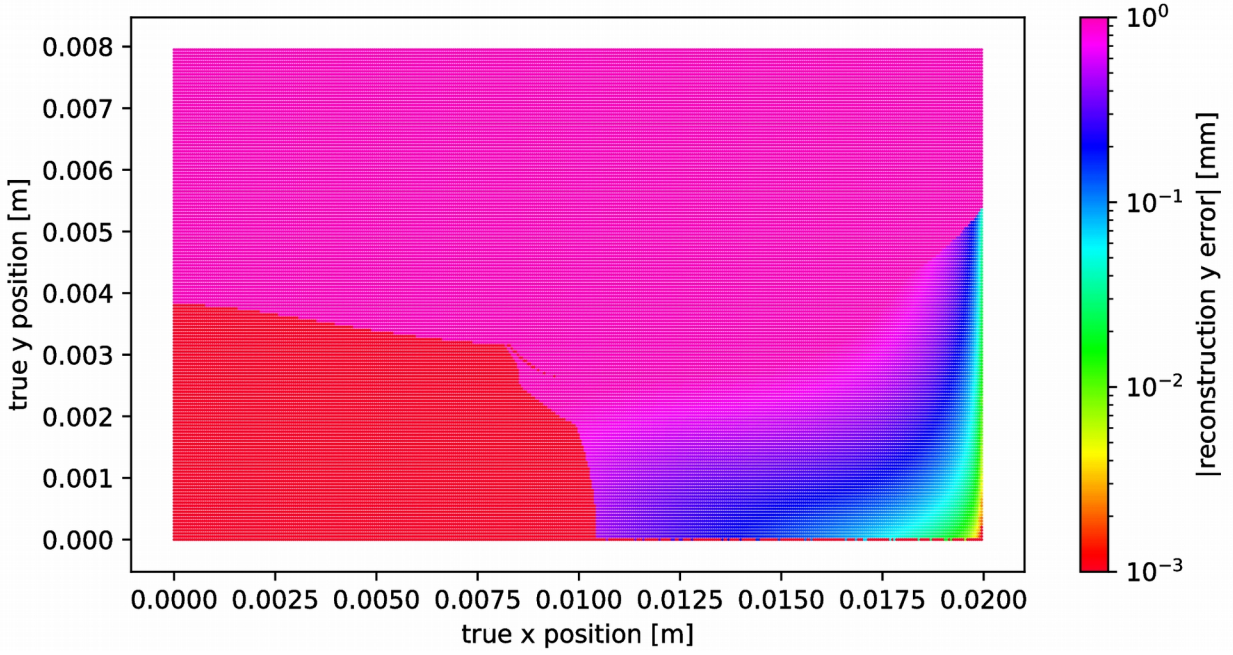
Dropping button 4

color bar clipped [1 um, 1 mm]

bpm_chessu_xyp.txt



average solving time = 1.0 ms



50 µm step size

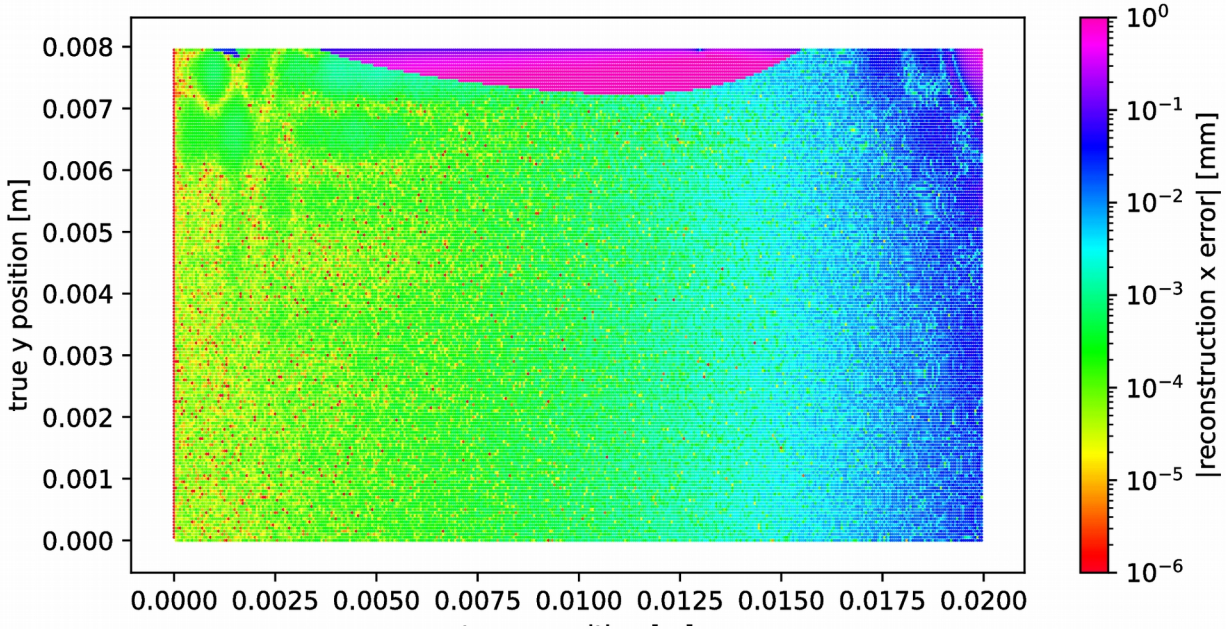
Additional materials

CESRV vs Python

color bar clipped [1 nm, 1 mm]

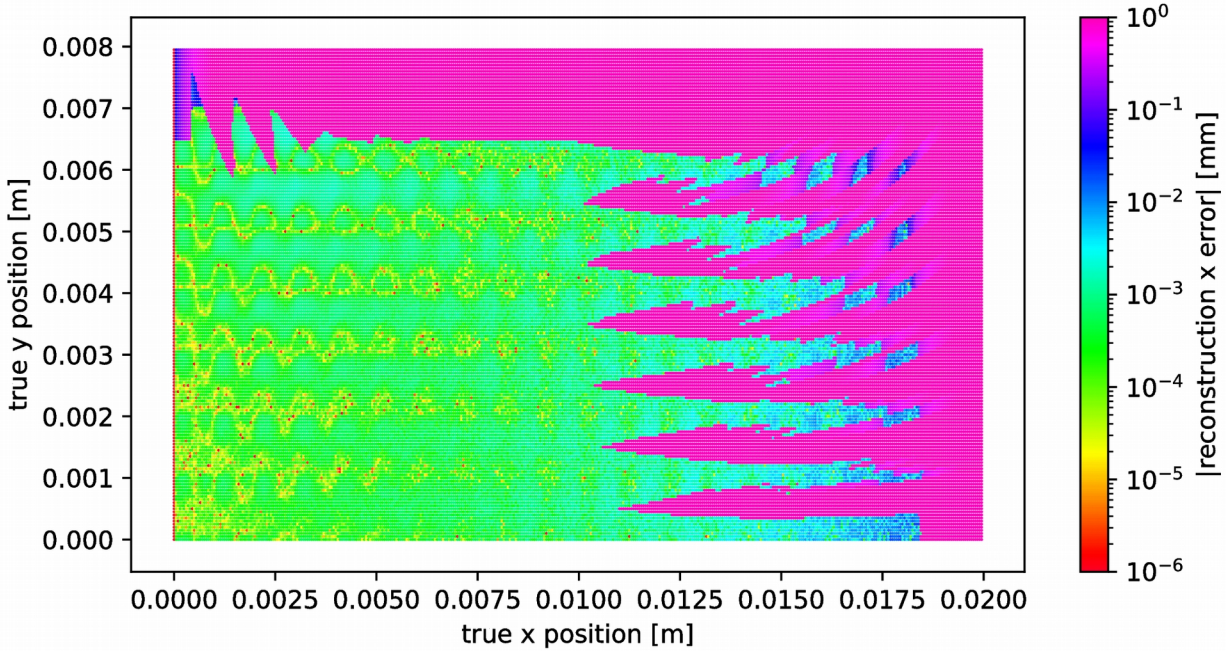
bpm_chessu_xyp.txt

Python



average solving time = 1.6 ms

CESRV



average solving time = 0.03 ms

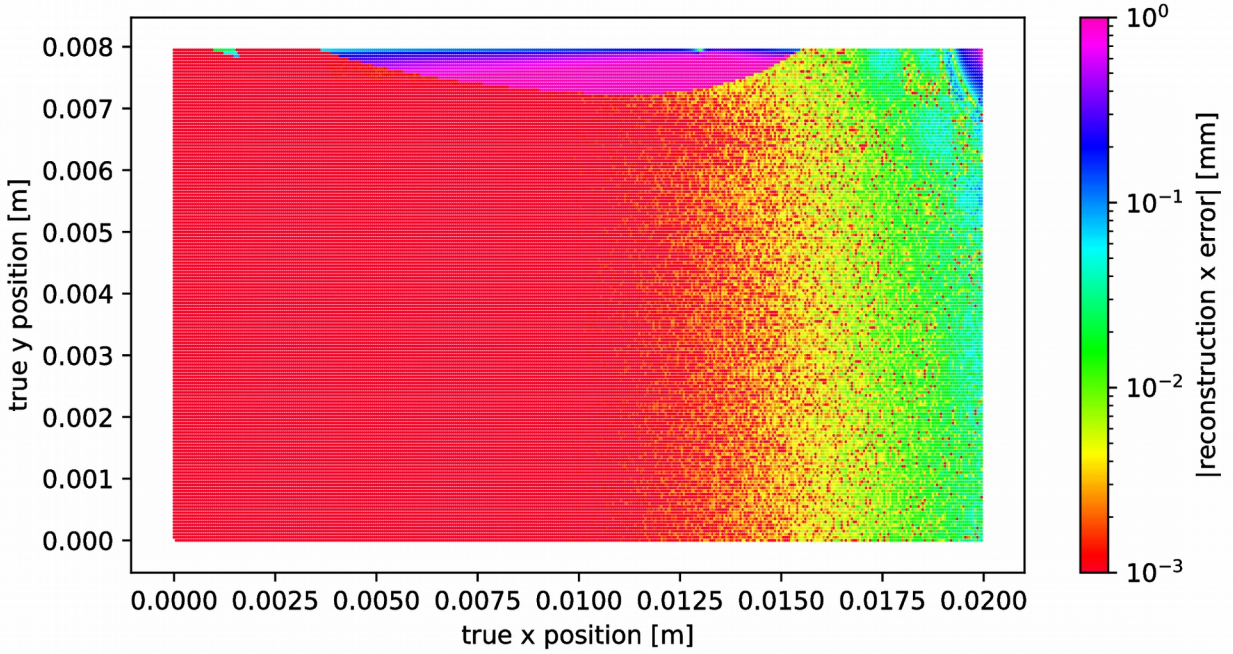
50 μm step size

CESRV vs Python

color bar clipped [1 um, 1 mm]

bpm_chessu_xyp.txt

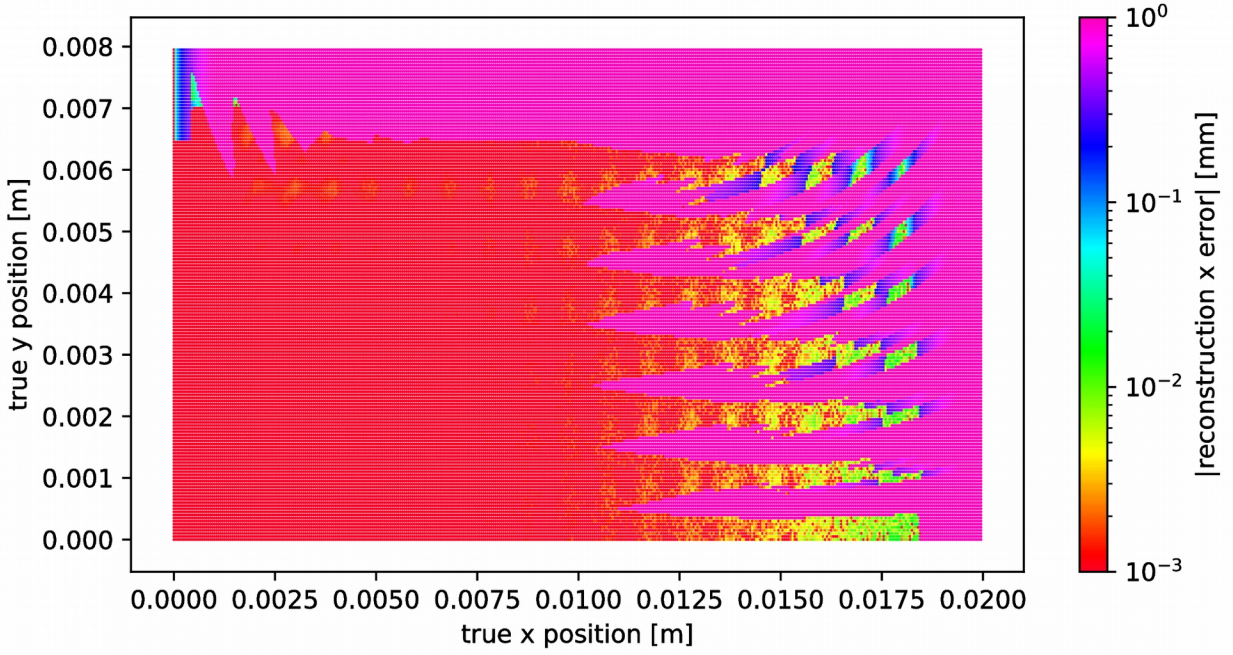
Python



average solving time = 1.6 ms

CESRV

(pink dots are mostly when minimization fails)



average solving time = 0.03 ms

50 μm step size