CBPM3 display

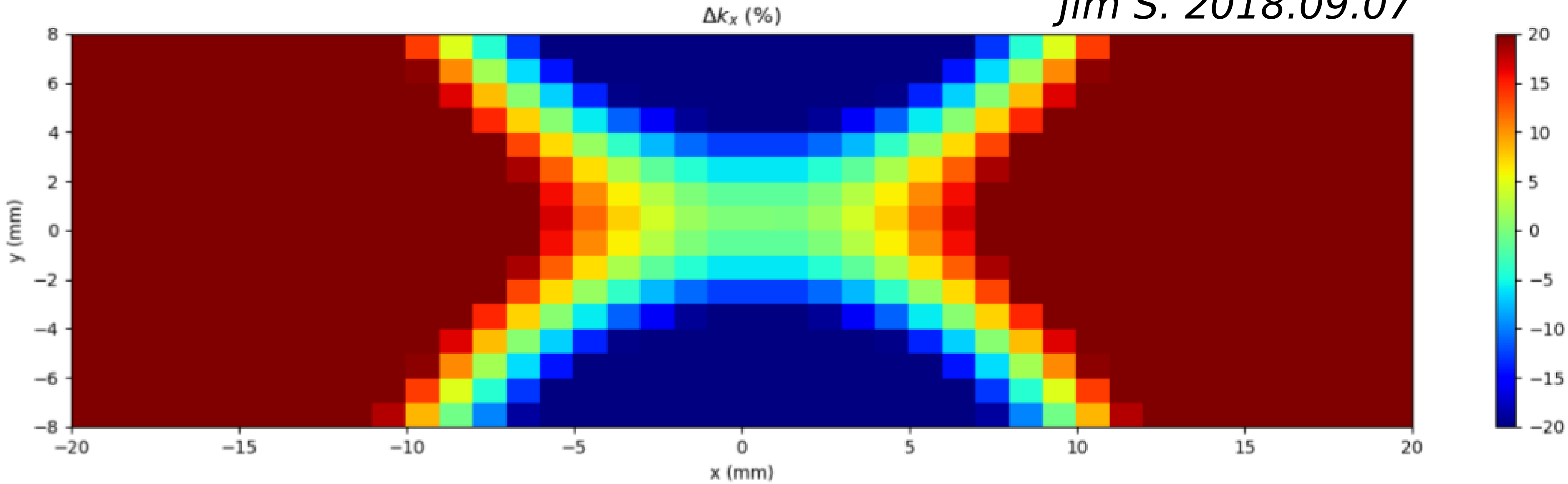
Antoine, Jim

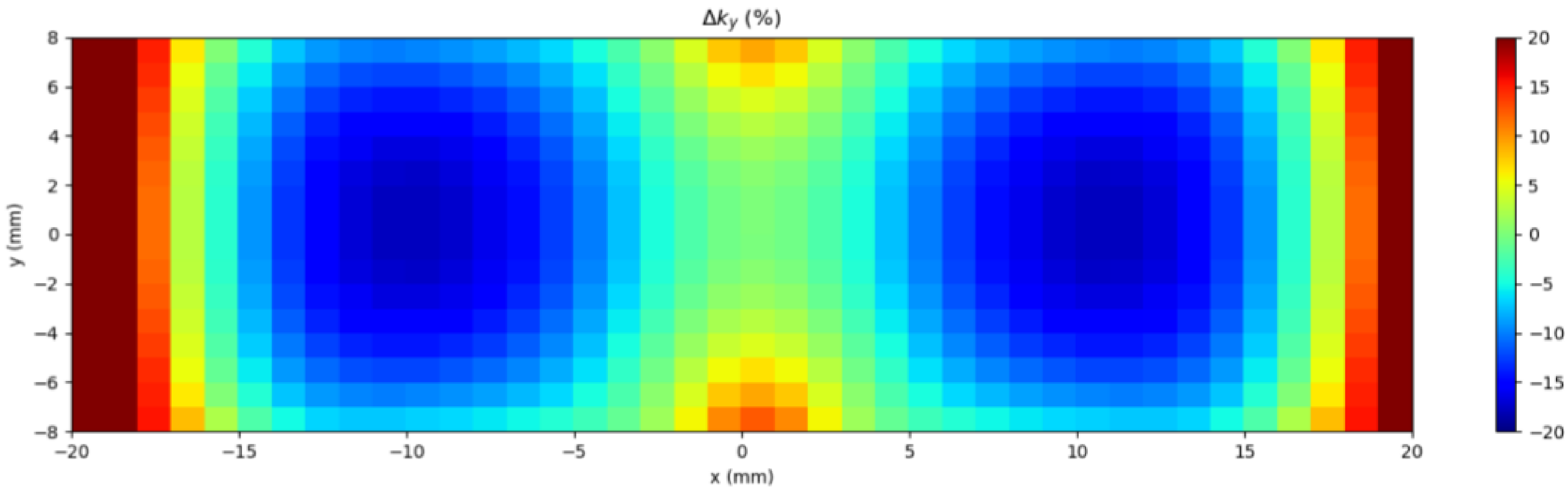
CBPM meeting

January 29, 2021

k_x and k_y deviation from linear regime (CHESS-U chamber):

Jim S. 2018.09.07

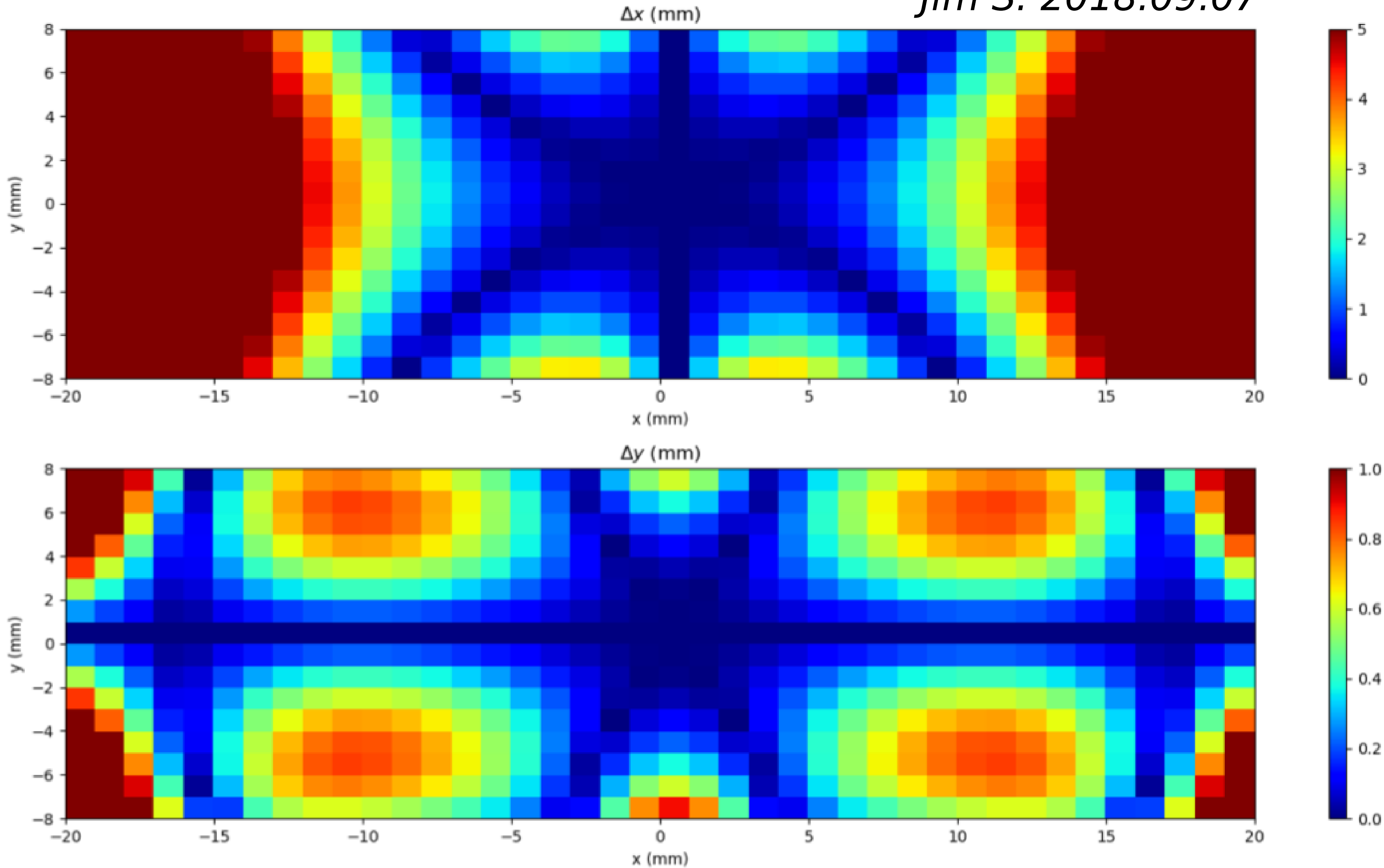




Non-linear (Poisson) beam position reconstruction

x and y deviation from linear regime (CHESS-U chamber):

Jim S. 2018.09.07



|Difference| between actual (x,y) and (x,y) as computed using linear $k_{x,y}$ from (x,y) = (0,0)

Poisson look-up table

Look-up table containing the non-linear response of button 1 as a function of static charge position (x,y)

/ [ACC] / CESR / CESR_instr / nonlin_bpm / nonlin_bpm_resource

Index of /CESR/CESR_instr/nonlin_bpm/nonlin_bpm_resource

Files shown: **9**
Directory revision: 42959 (of 49188)
Sticky Revision: [ ] Set

File ▲	Rev.	Age	Author	Last log entry
Parent Directory				
bpm_00_xyp.txt	**32754**	6 years	mjf7	Moving cesr-related programs out of /trunk/src
bpm_01_xyp.txt	**32754**	6 years	mjf7	Moving cesr-related programs out of /trunk/src
bpm_02_xyp.txt	**32754**	6 years	mjf7	Moving cesr-related programs out of /trunk/src
bpm_48_xyp.txt	**32754**	6 years	mjf7	Moving cesr-related programs out of /trunk/src
bpm_49_xyp.txt	**32754**	6 years	mjf7	Moving cesr-related programs out of /trunk/src
bpm_arc_xyp.txt	**32754**	6 years	mjf7	Moving cesr-related programs out of /trunk/src
bpm_chessu_xyp.txt	**42959**	2 years	js583	Update for CHESS-U BPM indexing; add new CHESS-U quad extrusion table. --JSh
button_coefficients.dat	**32754**	6 years	mjf7	Moving cesr-related programs out of /trunk/src
detcal.ok	**32754**	6 years	mjf7	Moving cesr-related programs out of /trunk/src

CESRV Fortran routines

Lots of routines, and many lines of code

/ [ACC] / CESR / CESR_instr / nonlin_bpm / code

Index of /CESR/CESR_instr/nonlin_bpm/code

Files shown: **11**
Directory revision: 44617 (of 49189)
Sticky Revision: [ ] Set

File ▲	Rev.	Age	Author	Last log entry
Parent Directory				
nonlin_bpm_init.f90	44599	21 months	dcs16	Remove debug printout.
nonlin_bpm_interpolate.f90	32754	6 years	mjf7	Moving cesr-related programs out of /trunk/src
nonlin_bpm_minimize.f90	32754	6 years	mjf7	Moving cesr-related programs out of /trunk/src
nonlin_bpm_mod.f90	43827	23 months	sw565	replace bpm calibration file detcal.ok with offset.bpm
nonlin_bpm_real_coords.f90	32754	6 years	mjf7	Moving cesr-related programs out of /trunk/src
nonlin_bpm_set_pointers.f90	44617	21 months	sw565	set 9W to be CHESS-U pipe BPM
nonlin_butcon.f90	32754	6 years	mjf7	Moving cesr-related programs out of /trunk/src
nonlin_mrqmin_mod.f90	32754	6 years	mjf7	Moving cesr-related programs out of /trunk/src
nonlin_orbit.f90	43527	2 years	sw565	fix detector mapping
nonlin_phase.f90	32754	6 years	mjf7	Moving cesr-related programs out of /trunk/src
nonlin_xy_shake_components.f90	44443	21 months	sw565	add comment to indicate cbpm idx

Question marks regarding what the code does and how/why, e.g.:

```fortran
  ! loop over buttons
  do k = 1, 4
     c = func_bpm%grid(ix,iy)%b(k)%c
     ! adapted from NR routine bcuint
     d(k,:,:) = 0.0
     ! evaluate polynomials with effecient but incomprehensible method
     do i = 4, 1, -1
        d(k,0,0)=t*d(k,0,0)+((c(i,4)*u+c(i,3))*u+c(i,2))*u+c(i,1)
        d(k,0,1)=t*d(k,0,1)+(3.0*c(i,4)*u+2.0*c(i,3))*u+c(i,2)
        d(k,0,2)=t*d(k,0,2)+(6.0*c(i,4)*u+2.0*c(i,3))
        d(k,1,0)=u*d(k,1,0)+(3.0*c(4,i)*t+2.0*c(3,i))*t+c(2,i)
        d(k,2,0)=u*d(k,2,0)+(6.0*c(4,i)*t+2.0*c(3,i))
     end do
     do i = 4, 2, -1
        d(k,1,1)=t*d(k,1,1)+(3.0*(i-1)*c(i,4)*u+2.0*(i-1)*c(i,3))*u+(i-1)*c(i,2)
     end do
  end do
  d(:,0,0) = d(:,0,0)
  d(:,1,0) = d(:,1,0)/(func_bpm%x(ix+1)-func_bpm%x(ix))
  d(:,2,0) = d(:,2,0)/(func_bpm%x(ix+1)-func_bpm%x(ix))**2
  d(:,0,1) = d(:,0,1)/(func_bpm%y(iy+1)-func_bpm%y(iy))
  d(:,0,2) = d(:,0,2)/(func_bpm%y(iy+1)-func_bpm%y(iy))**2
  d(:,1,1) = d(:,1,1)/((func_bpm%x(ix+1)-func_bpm%x(ix)) * &
       (func_bpm%y(iy+1)-func_bpm%y(iy)))

  d = d * x(3)
end subroutine nonlin_bpm_interpolate
```

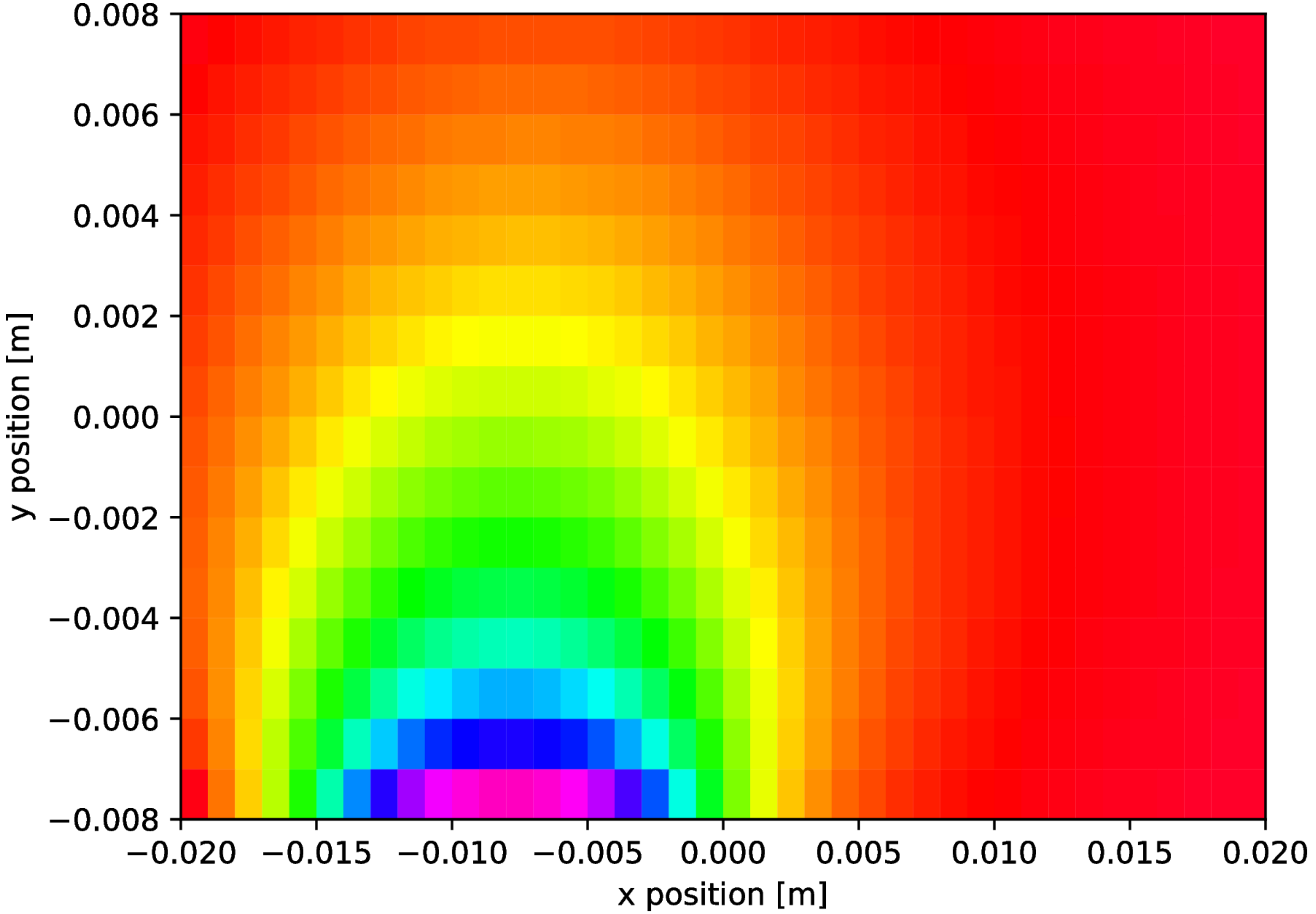
Python non-linear code

Instead of copying/porting Fortran code: write Python code from "first principles" hoping for clearer, shorter, faster code

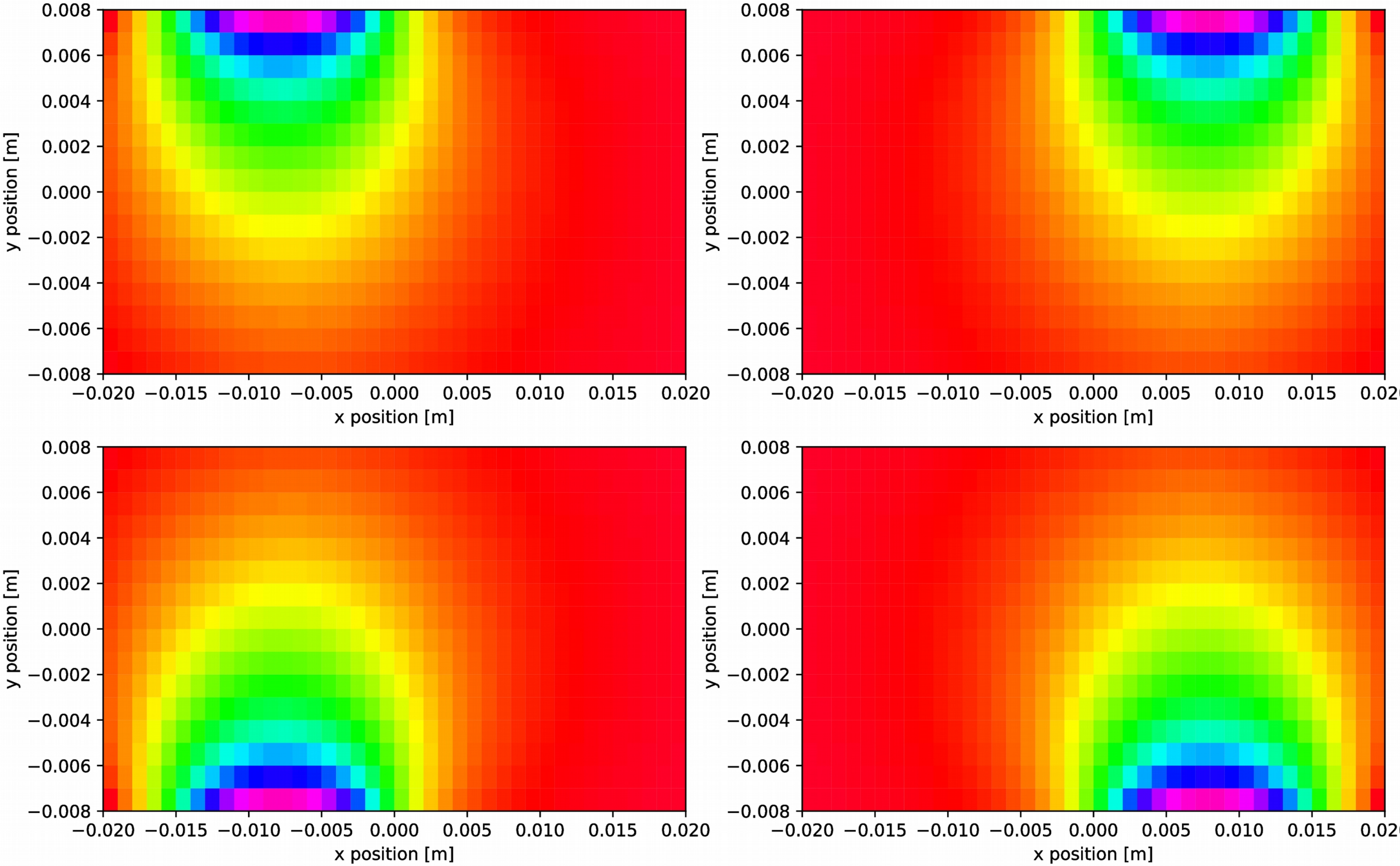
The procedure is understood:

- ✗ fetch button 1 look-up table and create tables for button 2, 3, 4 via reflections
- ✗ apply normalization to look-up tables
- ✗ generate fine map interpolating tables
- ✗ minimize merit function to find (x,y) corresponding to measured b1, b2, b3, b4

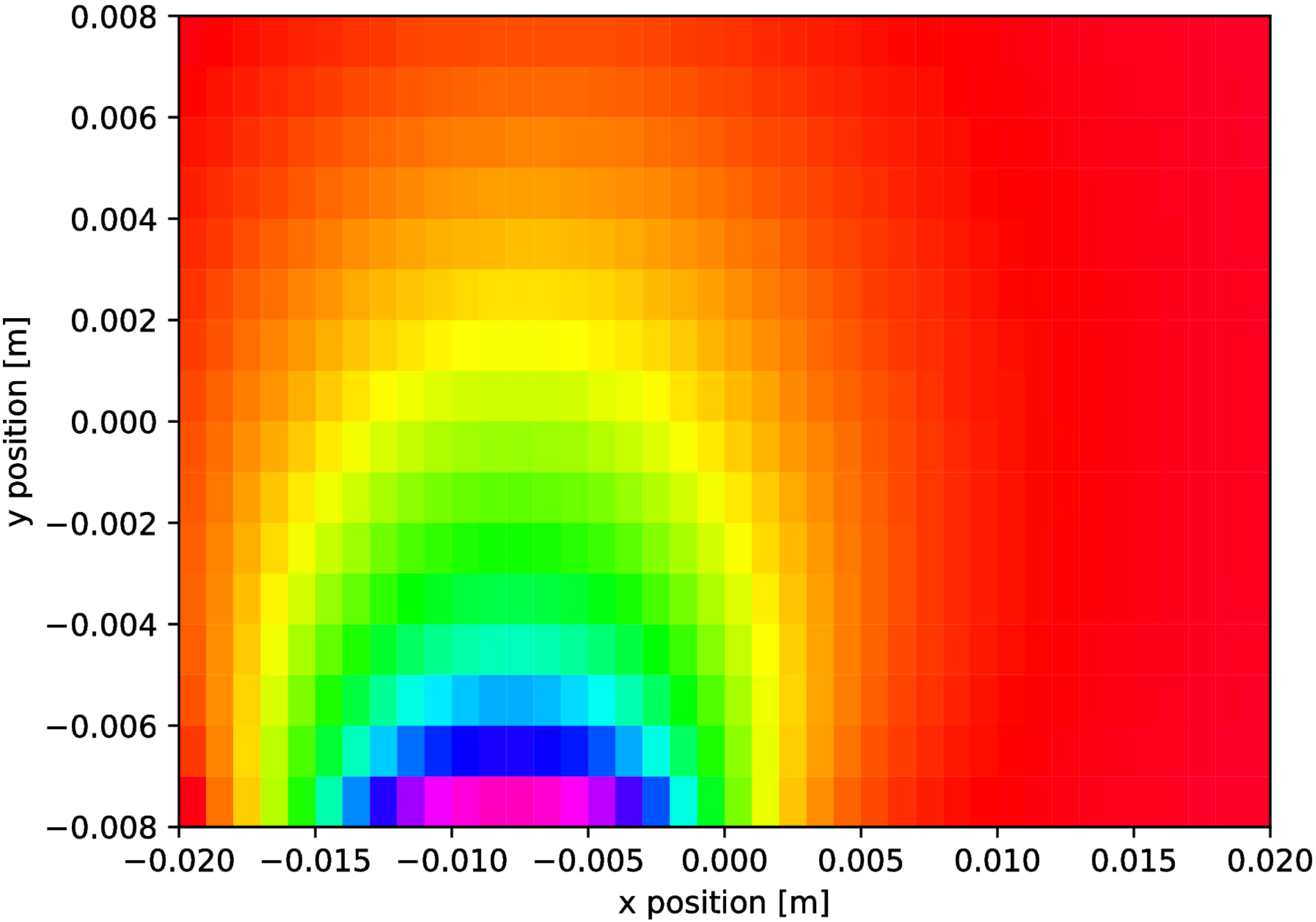
Read-in and plot *bpm_chessu_xyp.txt* button 1 response:



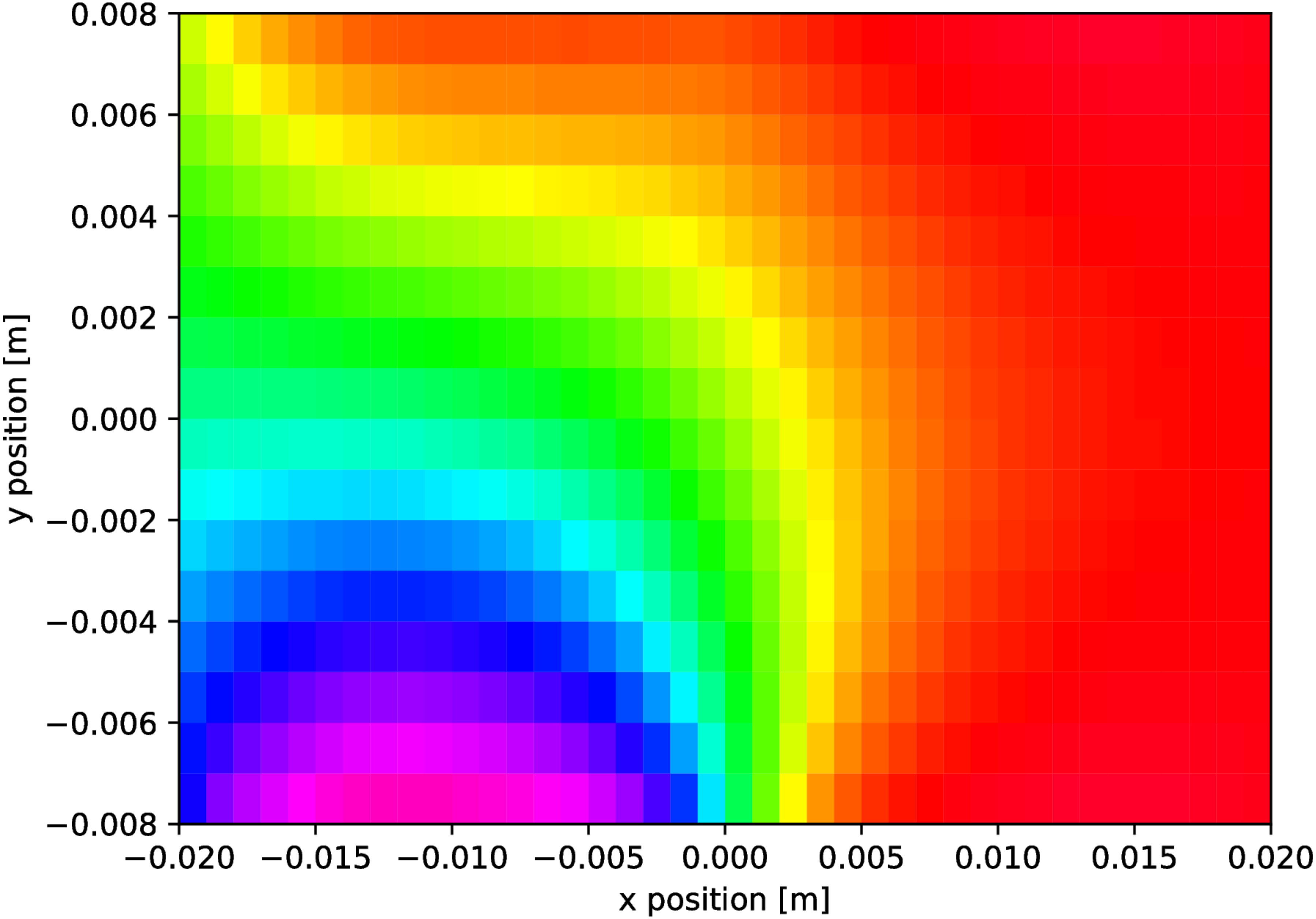
Look-up tables: reflect b1 → b2, b3, b4



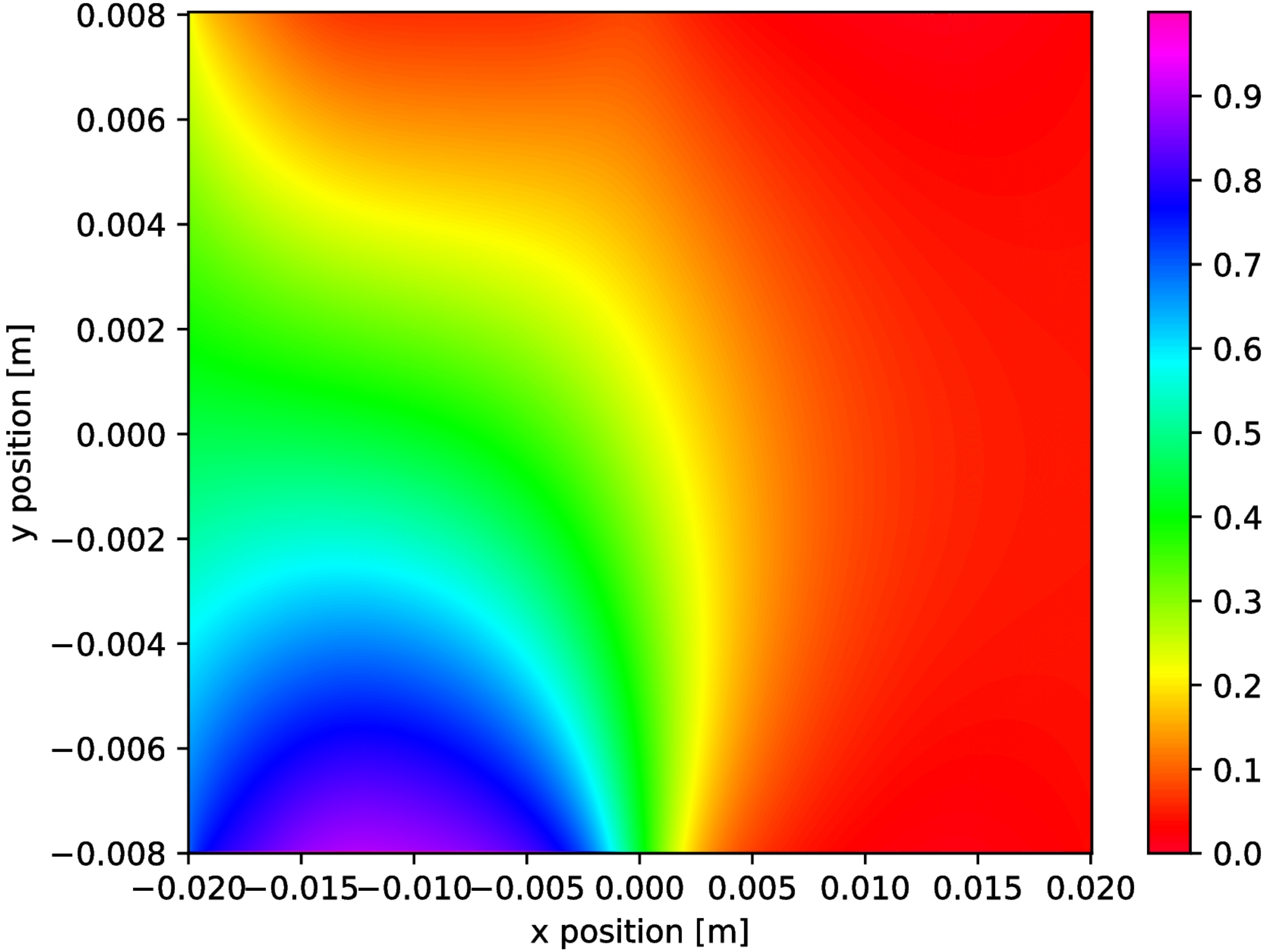
Button 1 **before** normalization



Button 1 **after** normalization



Quintic (order 5) 2D interpolation:



Minimization to extract (x,y)

The goal is to match normalized measured amplitudes to normalized look-up table amplitudes to extract best (x,y). One merit function to minimize can be:

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

sum over 4 buttons

measured i^{th} button amplitude

2D interpolation of look-up table

two unknowns

Sum of square is required to avoid cancellation between different buttons as we want to minimize the difference between measured/look-up for all the buttons. The overall square root is used to go back to the “natural” unit of button amplitude.

A technicality

No need to create the map for buttons 2, 3 and 4 → can use the interpolated map from button 1 to do it all and simplify the code:

```
def f_merit_2(params, *args):
    x, y = params
    return np.sqrt(
                    (args[0]-f_norm_b1(x, y))**2 + #  button 1
                    (args[1]-f_norm_b1(-1*x, y))**2 +  # button 2
                    (args[2]-f_norm_b1(x, -1*y))**2 +  # button 3
                    (args[3]-f_norm_b1(-1*x, -1*y))**2  # button 4
                )
```

It was checked that using only button 1's map versus using all the maps yield the exact same results (as expected)

Exact solution as initial guess

Case 1:

✗ initial guess = exact (x, y) solutions

✗ bound minimization to entire 2D space (x),(y)=(-0.02,0.02),(-0.008,0.008)

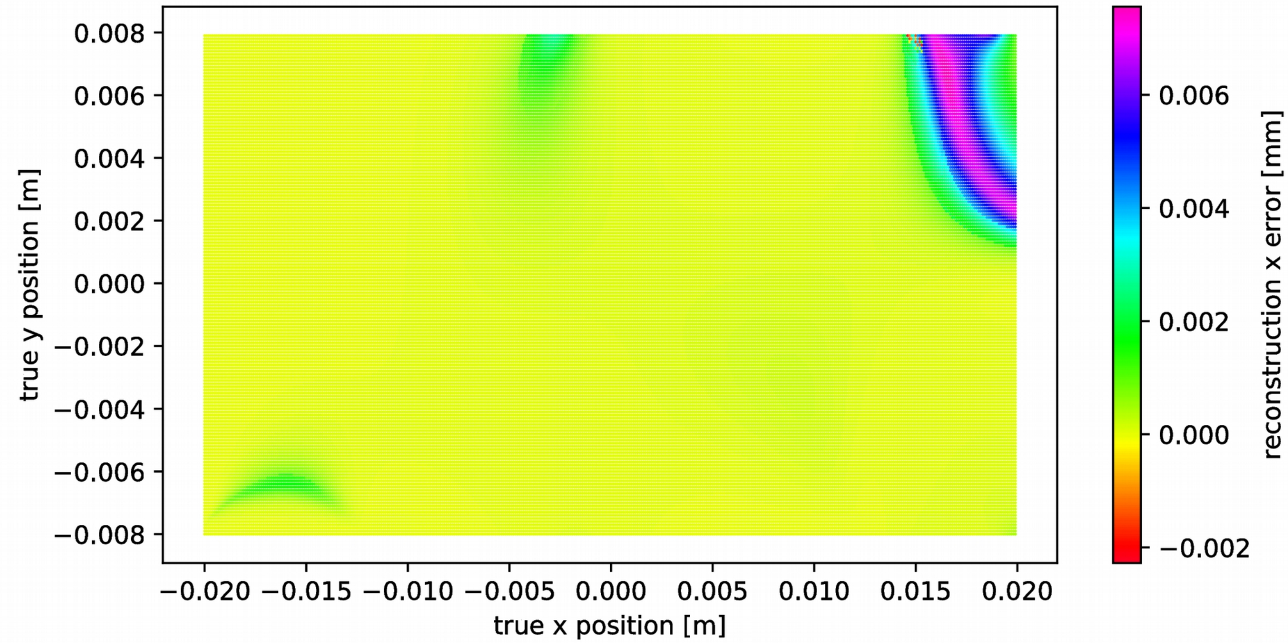
✗ default minimizer configurations

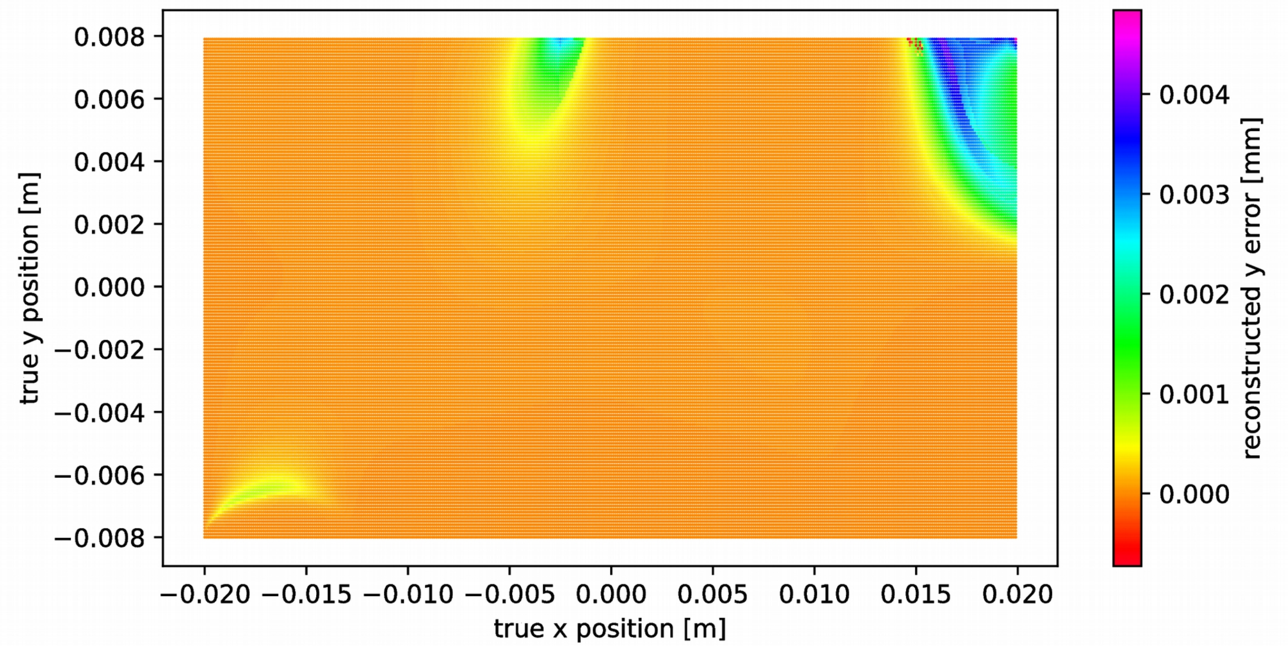
```
def pos(params, *args):
    x, y = params
    return np.sqrt(
                    (args[0]-f_norm_b1(x, y))**2 + #  button 1
                    (args[1]-f_norm_b1(-1*x, y))**2 +  # button 2
                    (args[2]-f_norm_b1(x, -1*y))**2 +  # button 3
                    (args[3]-f_norm_b1(-1*x, -1*y))**2  # button 4
                )
```

```
result = optimize.minimize(
                            pos,
                            initial_guess,
                            args=(b1, b2, b3, b4),
                            method='SLSQP',
                            bounds=((-0.02, 0.02), (-0.008, 0.008))
                        )
```

bpm_chessu_xyp.txt

Case 1:





Minimization to extract (x,y)

Case 2:

✗ initial guess = exact (x, y) solutions * 1.1 (i.e. +10% offset)

✗ bound minimization to entire 2D space (x),(y)=(-0.02,0.02),(-0.008,0.008)

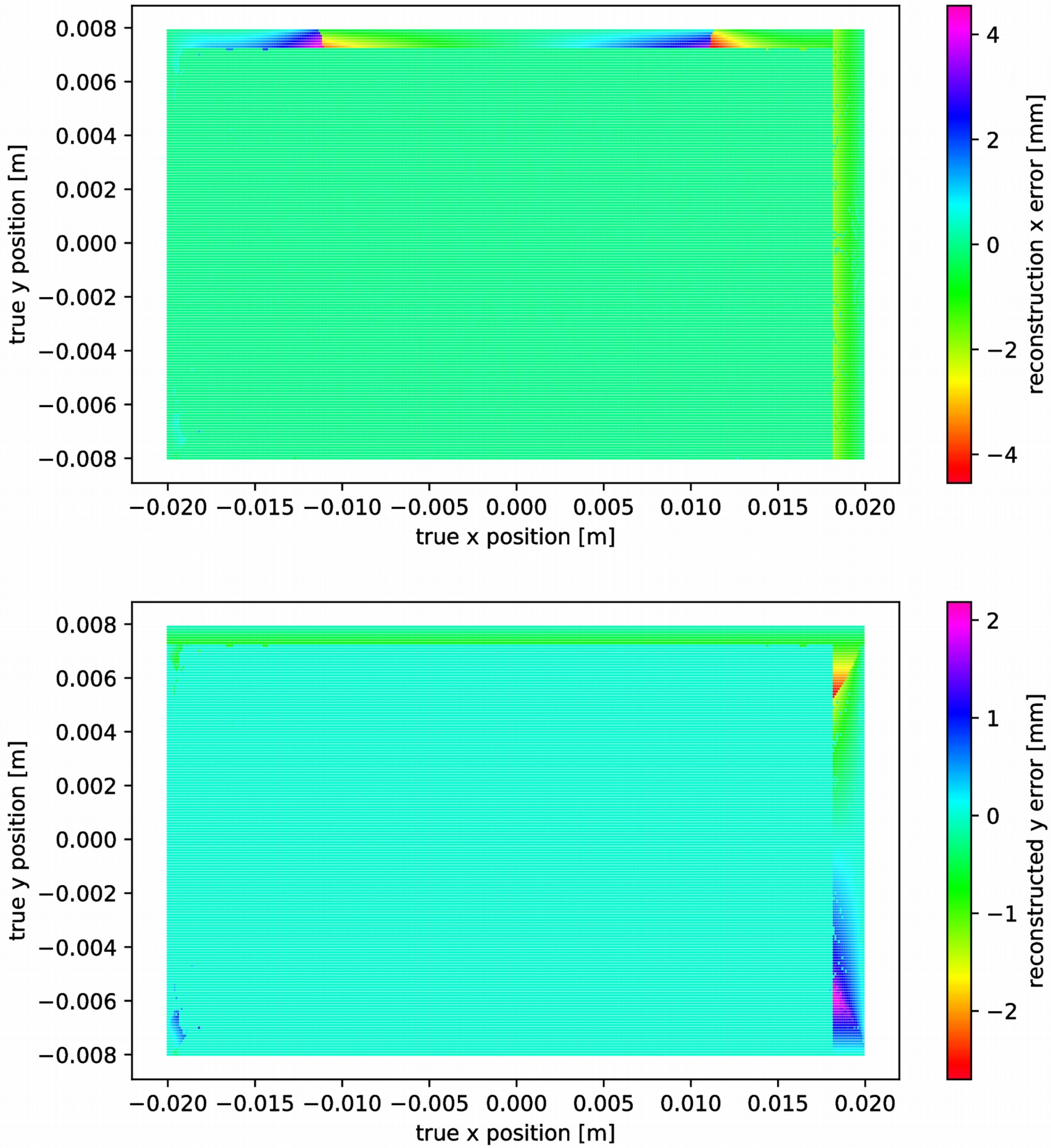
✗ default minimizer configurations

```
def pos(params, *args):
    x, y = params
    return np.sqrt(
                    (args[0]-f_norm_b1(x, y))**2 + #  button 1
                    (args[1]-f_norm_b1(-1*x, y))**2 +  # button 2
                    (args[2]-f_norm_b1(x, -1*y))**2 +  # button 3
                    (args[3]-f_norm_b1(-1*x, -1*y))**2  # button 4
                )
```

```
result = optimize.minimize(
                            pos,
                            initial_guess,
                            args=(b1, b2, b3, b4),
                            method='SLSQP',
                            bounds=((-0.02, 0.02), (-0.008, 0.008))
                        )
```

bpm_chessu_xyp.txt

Case 2:



Case 3:

- ✗ initial guess = exact (x, y) solutions * 1.3 (i.e. +30% offset)
- ✗ bound minimization to entire 2D space (x),(y)=(-0.02,0.02),(-0.008,0.008)
- ✗ default minimizer configurations

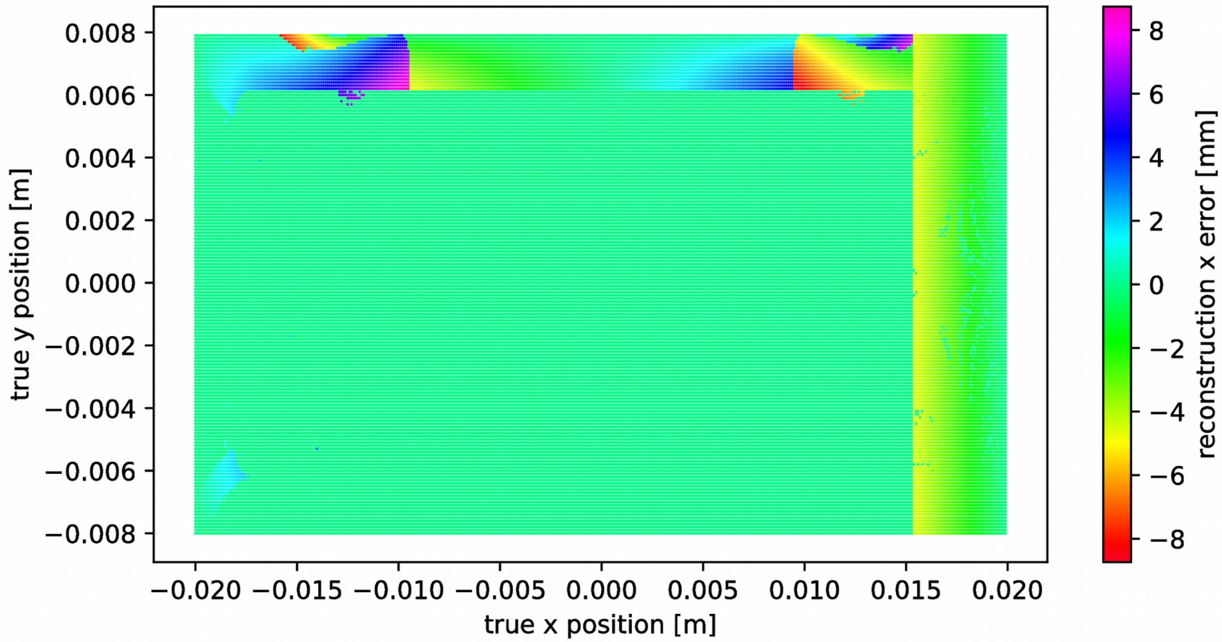
```
def pos(params, *args):
    x, y = params
    return np.sqrt(
                    (args[0]-f_norm_b1(x, y))**2 + #  button 1
                    (args[1]-f_norm_b1(-1*x, y))**2 +  # button 2
                    (args[2]-f_norm_b1(x, -1*y))**2 +  # button 3
                    (args[3]-f_norm_b1(-1*x, -1*y))**2  # button 4
                )
```

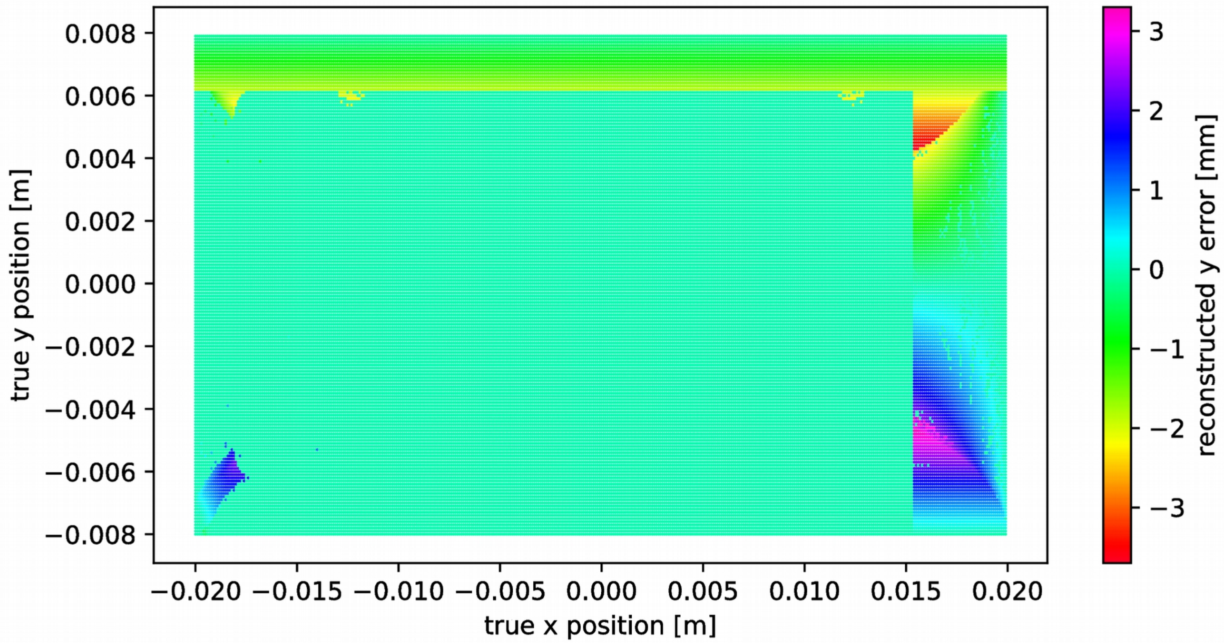
```
result = optimize.minimize(
                            pos,
                            initial_guess,
                            args=(b1, b2, b3, b4),
                            method='SLSQP',
                            bounds=((-0.02, 0.02), (-0.008, 0.008))
                        )
```

Minimization to extract (x,y)

bpm_chessu_xyp.txt

Case 3:





k_x, k_y as initial guess

Minimization to extract (x,y)

Case 4:

✗ initial guess = positions computed using k_x=0.0102 m and k_y=0.0104 m

✗ bound minimization to entire 2D space (x),(y)=(-0.02,0.02),(-0.008,0.008)

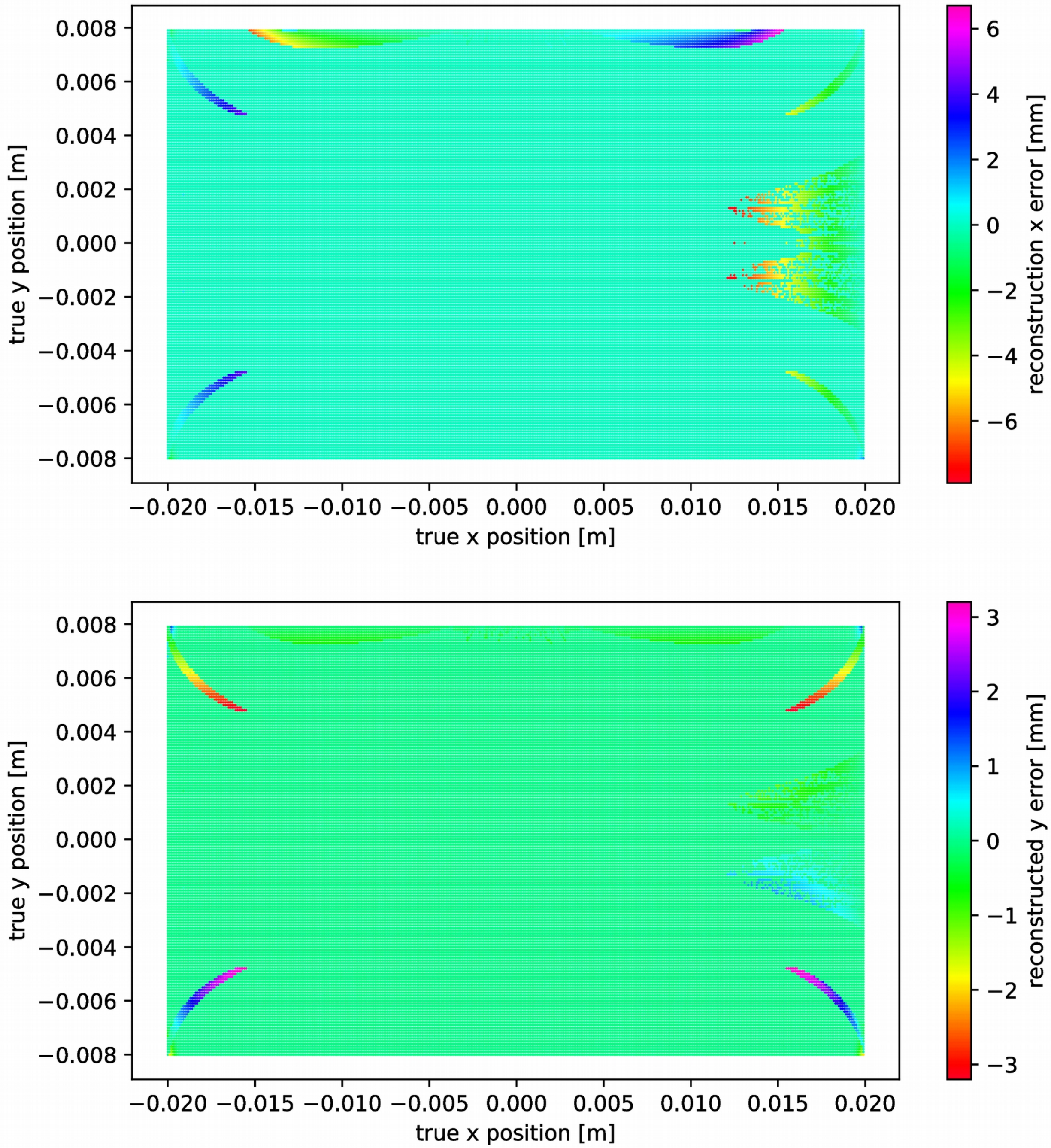
✗ default minimizer configurations

```
def pos(params, *args):
    x, y = params
    return np.sqrt(
                    (args[0]-f_norm_b1(x, y))**2 + #  button 1
                    (args[1]-f_norm_b1(-1*x, y))**2 +  # button 2
                    (args[2]-f_norm_b1(x, -1*y))**2 +  # button 3
                    (args[3]-f_norm_b1(-1*x, -1*y))**2  # button 4
                )
```

```
result = optimize.minimize(
                            pos,
                            initial_guess,
                            args=(b1, b2, b3, b4),
                            method='SLSQP',
                            bounds=((-0.02, 0.02), (-0.008, 0.008))
                        )
```

bpm_chessu_xyp.txt

Case 4:



Initial guess from look-up table

Case 5:

✗ initial guess = minimize merit function using raw look-up table

✗ bound minimization to entire 2D space (x),(y)=(-0.02,0.02),(-0.008,0.008)

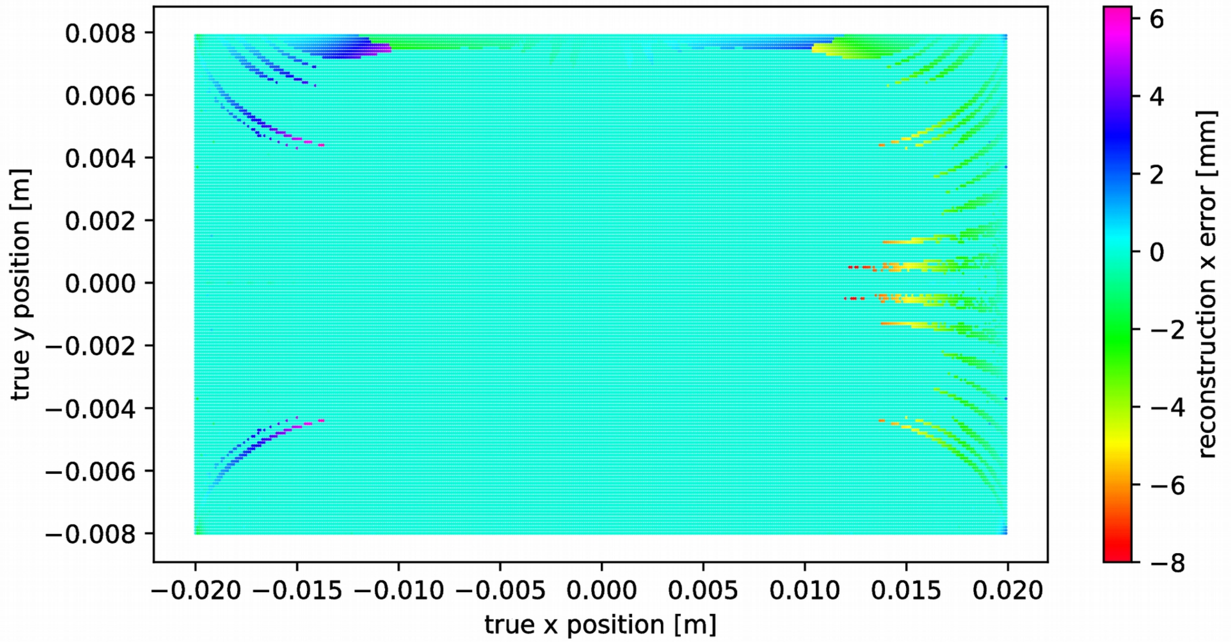
✗ default minimizer configurations

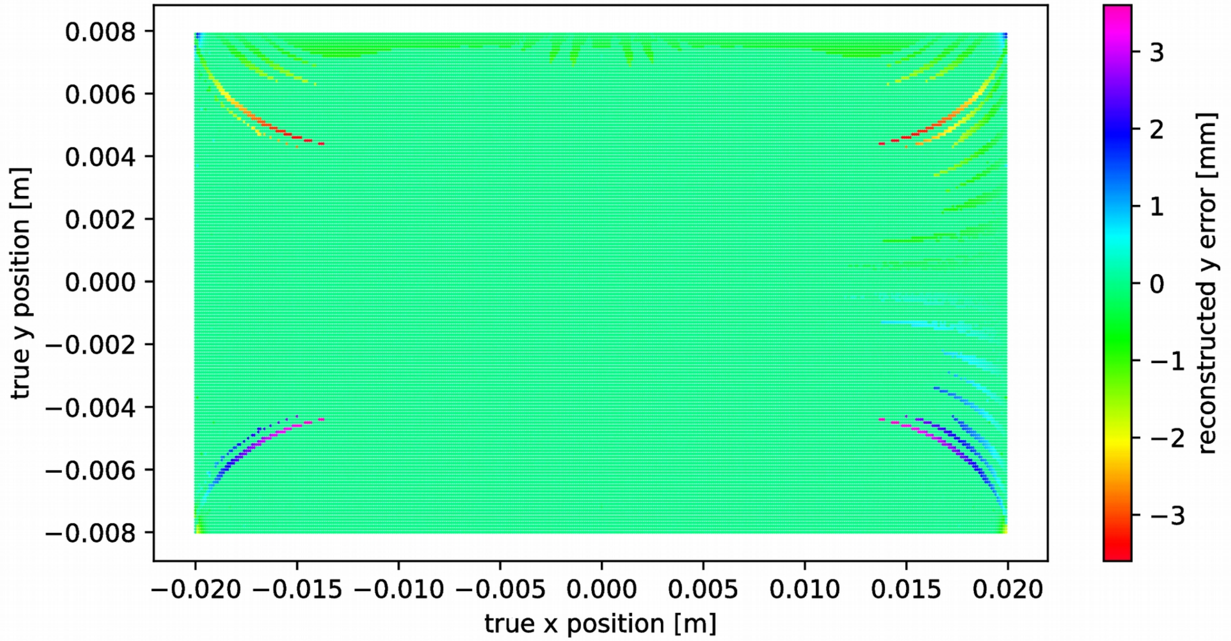
```
def pos(params, *args):
    x, y = params
    return np.sqrt(
                    (args[0]-f_norm_b1(x, y))**2 + #  button 1
                    (args[1]-f_norm_b1(-1*x, y))**2 +  # button 2
                    (args[2]-f_norm_b1(x, -1*y))**2 +  # button 3
                    (args[3]-f_norm_b1(-1*x, -1*y))**2  # button 4
                )
```

```
result = optimize.minimize(
                            pos,
                            initial_guess,
                            args=(b1, b2, b3, b4),
                            method='SLSQP',
                            bounds=((-0.02, 0.02), (-0.008, 0.008))
                        )
```

bpm_chessu_xyp.txt

Case 5:



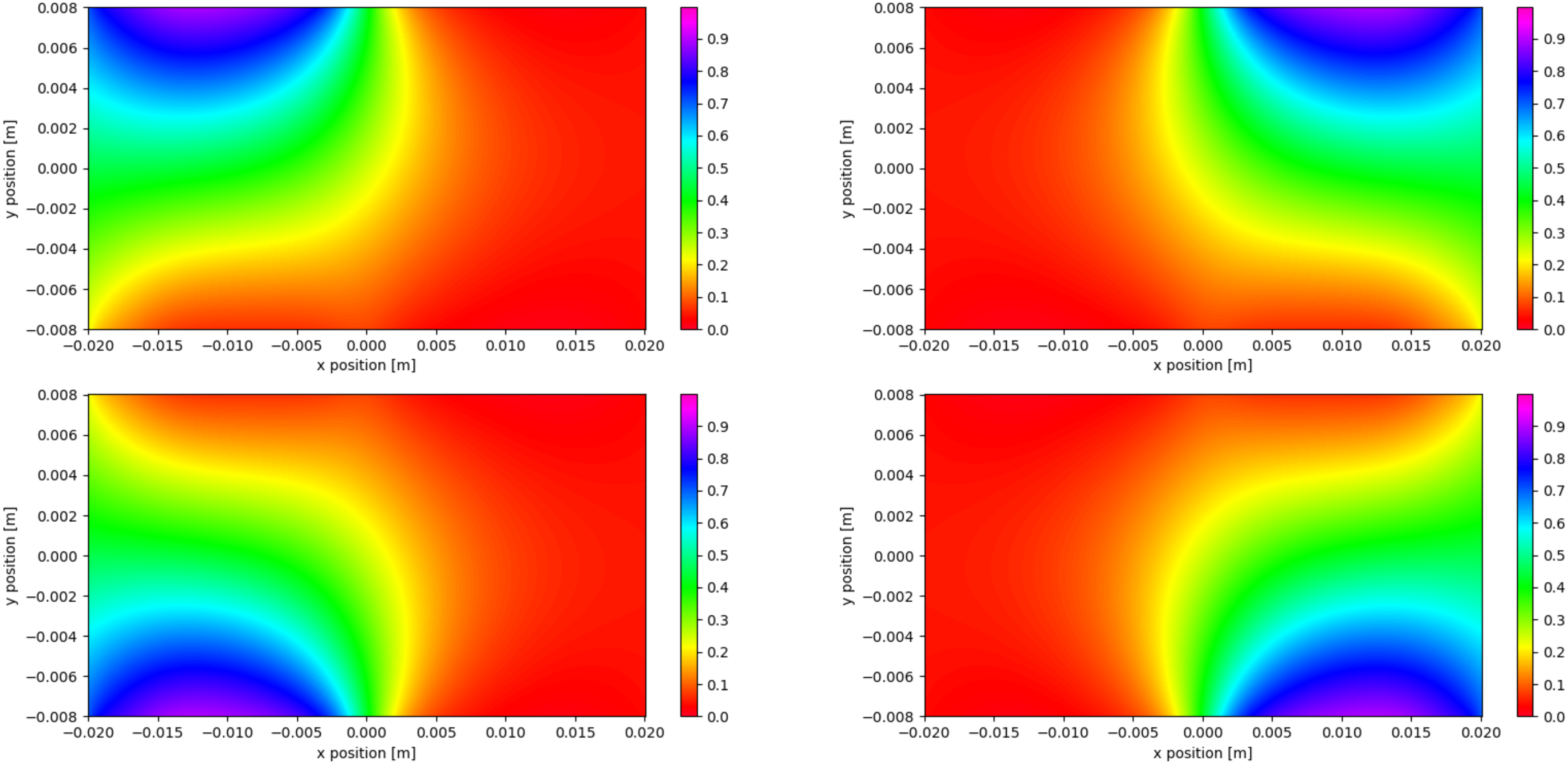


New merit function

New merit function

The current merit function could be struggling with equipotentials at large excursion since it deals with absolute differences:

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



New merit function

Let’s say button 1 (bottom left) reads a relative amplitude of about 0.75. There is an equipotential on button 1’s map allowing for many (x, y) pairs

Button 3 (top left) constrains a bit the equipotential as its relative amplitude along the line varies from 0 to 0.2

Button 2 and 4 reads amplitude of about 0 and therefore any deviation from the true (x, y) will yield small absolute differences since dealing with small numbers → do not help the merit function much

An alternative merit function is to use relative differences so each button can contribute significantly:

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

Case 6:

✗ initial guess = minimize merit function using raw look-up table

✗ bound minimization to entire 2D space (x),(y)=(-0.02,0.02),(-0.008,0.008)

✗ default minimizer configurations

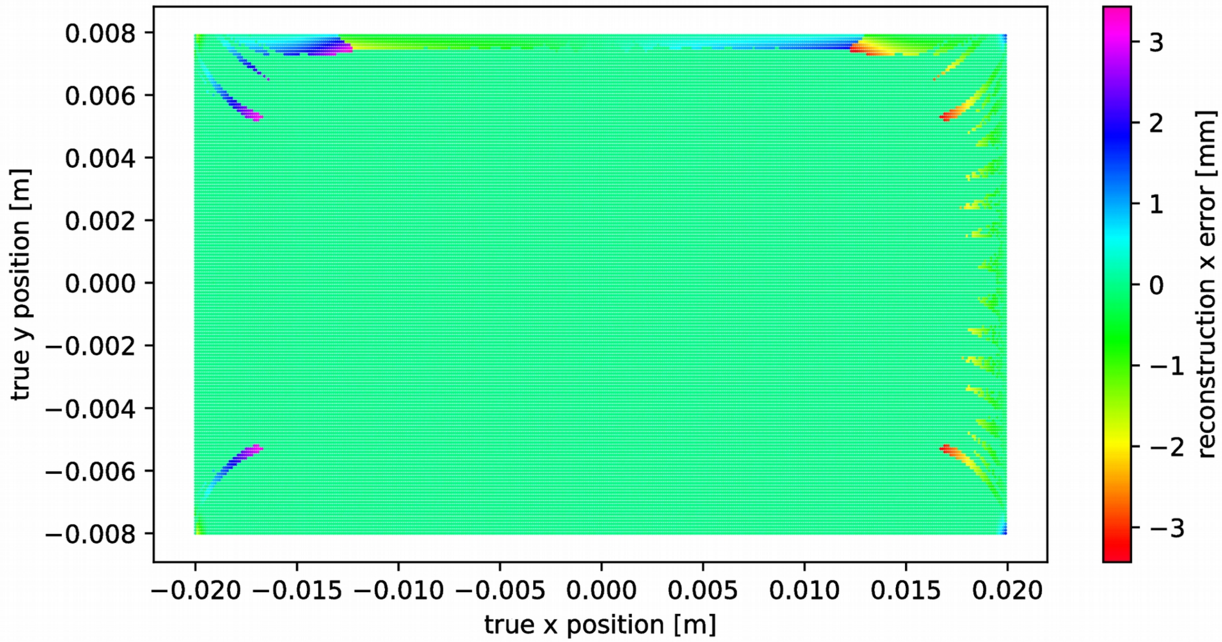
```
def pos(params, *args):
    x, y = params
    return np.sqrt(
                    (args[0]-f_norm_b1(x, y))**2 + #  button 1
                    (args[1]-f_norm_b1(-1*x, y))**2 +  # button 2
                    (args[2]-f_norm_b1(x, -1*y))**2 +  # button 3
                    (args[3]-f_norm_b1(-1*x, -1*y))**2  # button 4
                )
```

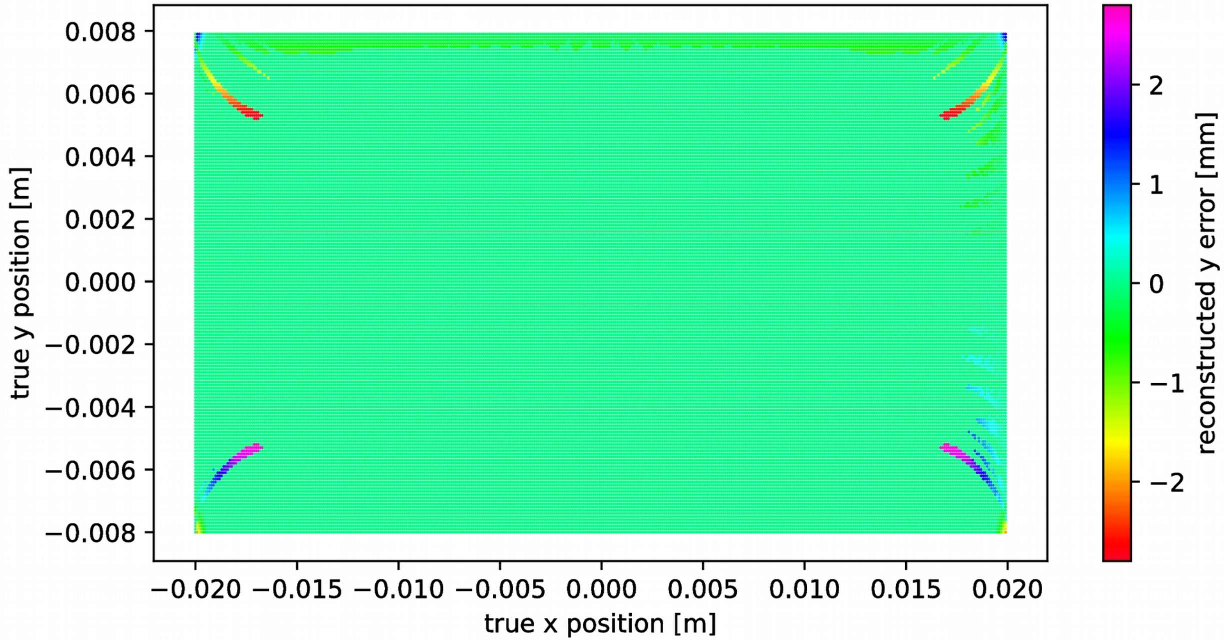
```
result = optimize.minimize(
                            pos,
                            initial_guess,
                            args=(b1, b2, b3, b4),
                            method='SLSQP',
                            bounds=((-0.02, 0.02), (-0.008, 0.008))
                        )
```

Minimization to extract (x,y)

bpm_chessu_xyp.txt

Case 6:

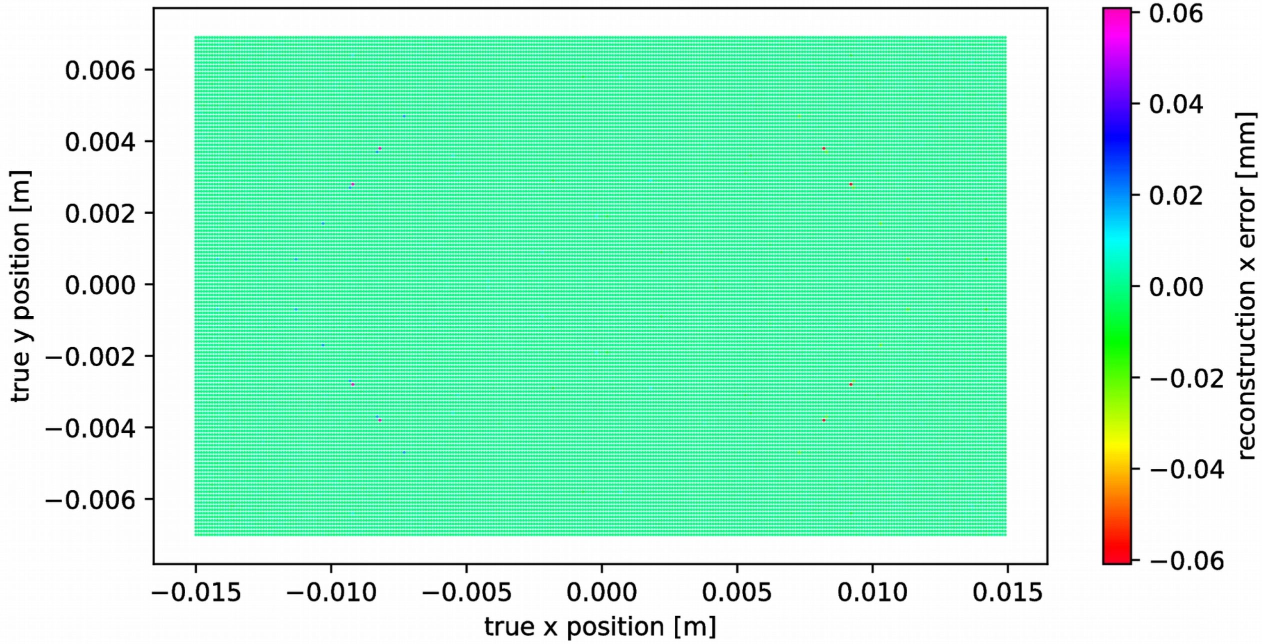


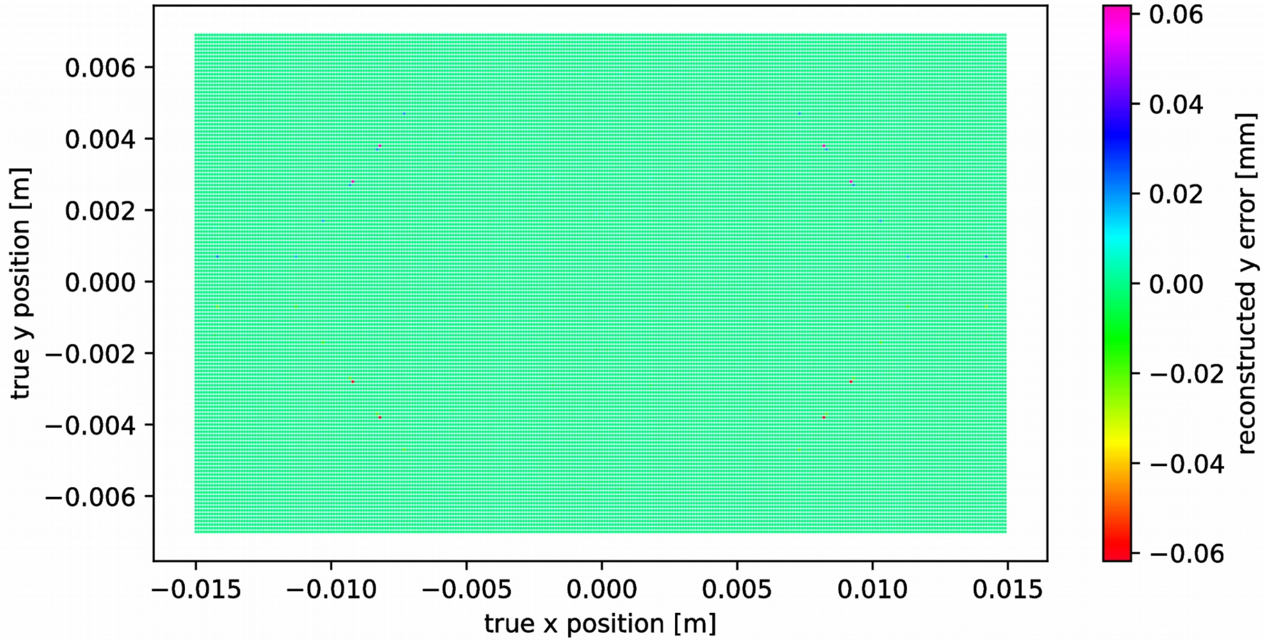


Minimization to extract (x,y)

:ssu_xyp.txt

Case 6:





Minimization to extract (x,y)

Case 7:

✗ initial guess = minimize merit function using raw look-up table

✗ bound minimization to entire 2D space (x),(y)=(-0.02,0.02),(-0.008,0.008)

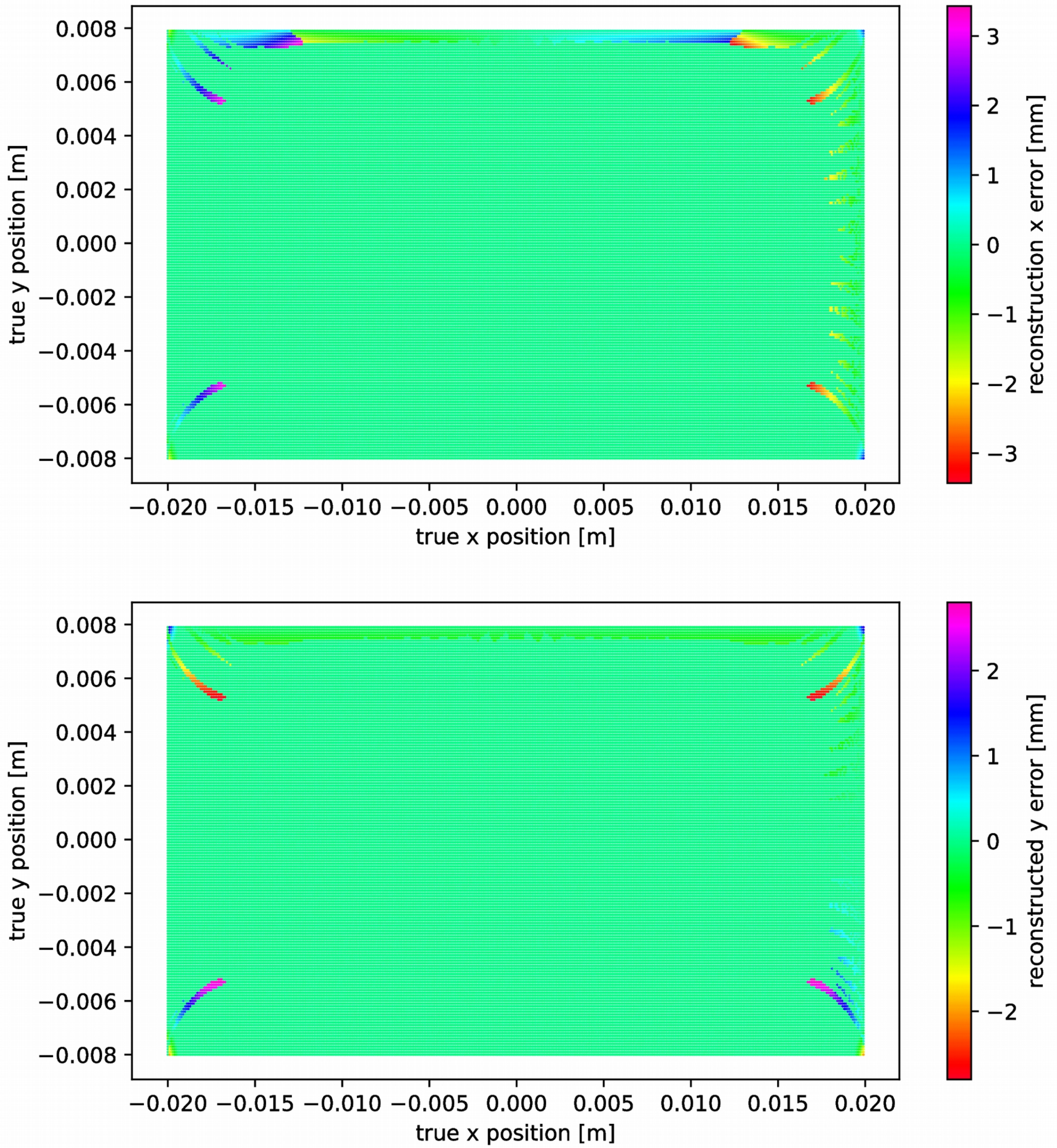
✗ tweaked minimizer configurations

```
def pos(params, *args):
    x, y = params
    return np.sqrt(
                    (args[0]-f_norm_b1(x, y))**2 + #  button 1
                    (args[1]-f_norm_b1(-1*x, y))**2 +  # button 2
                    (args[2]-f_norm_b1(x, -1*y))**2 +  # button 3
                    (args[3]-f_norm_b1(-1*x, -1*y))**2  # button 4
                )
```

```
result = optimize.minimize(
                            f_merit_3,
                            initial_guess,
                            args=(button[0], button[1], button[2], button[3]),
                            method='SLSQP',
                            bounds=((-0.02,0.02),(-0.008,0.008)),
                            options={'maxiter': 5000, 'ftol':1e-8, 'eps': 1.e-10}
                        )
```

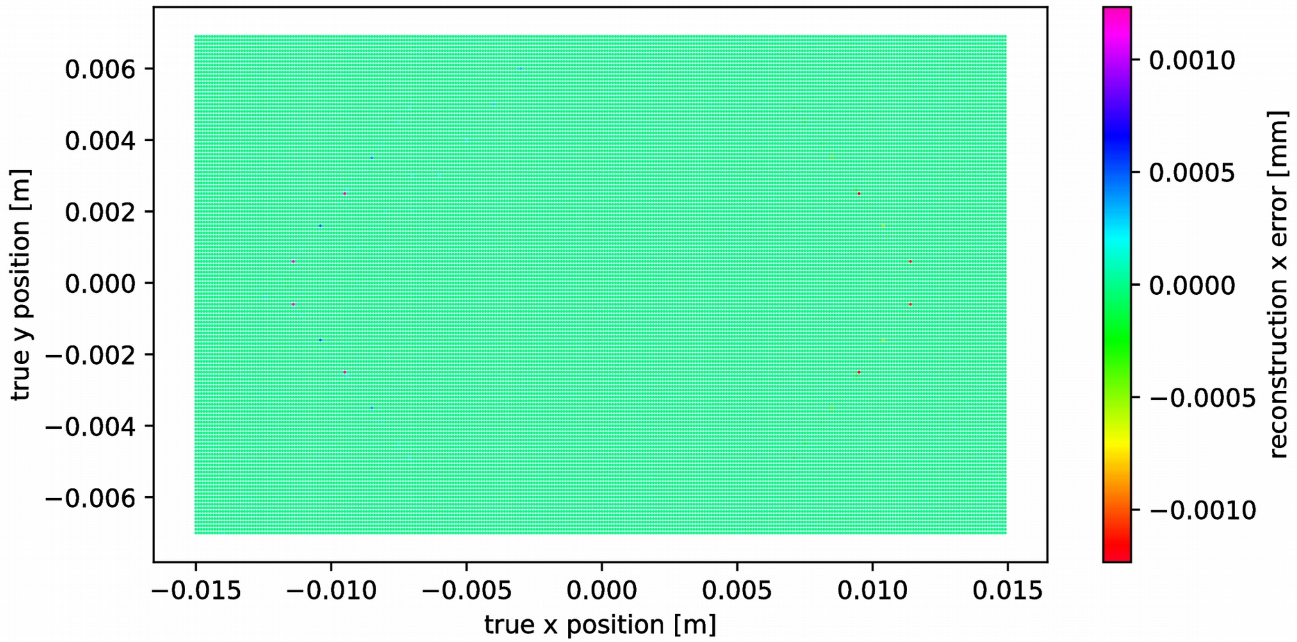
bpm_chessu_xyp.txt

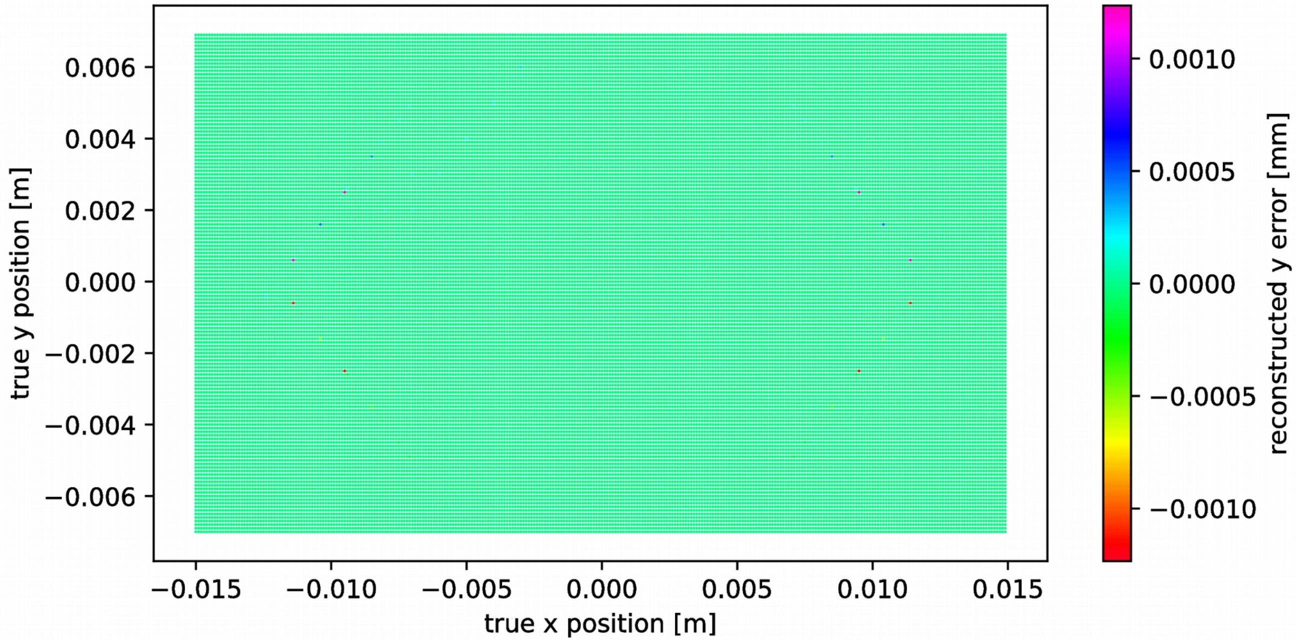
Case 7:



bpm_chessu_xyp.txt

Case 7:

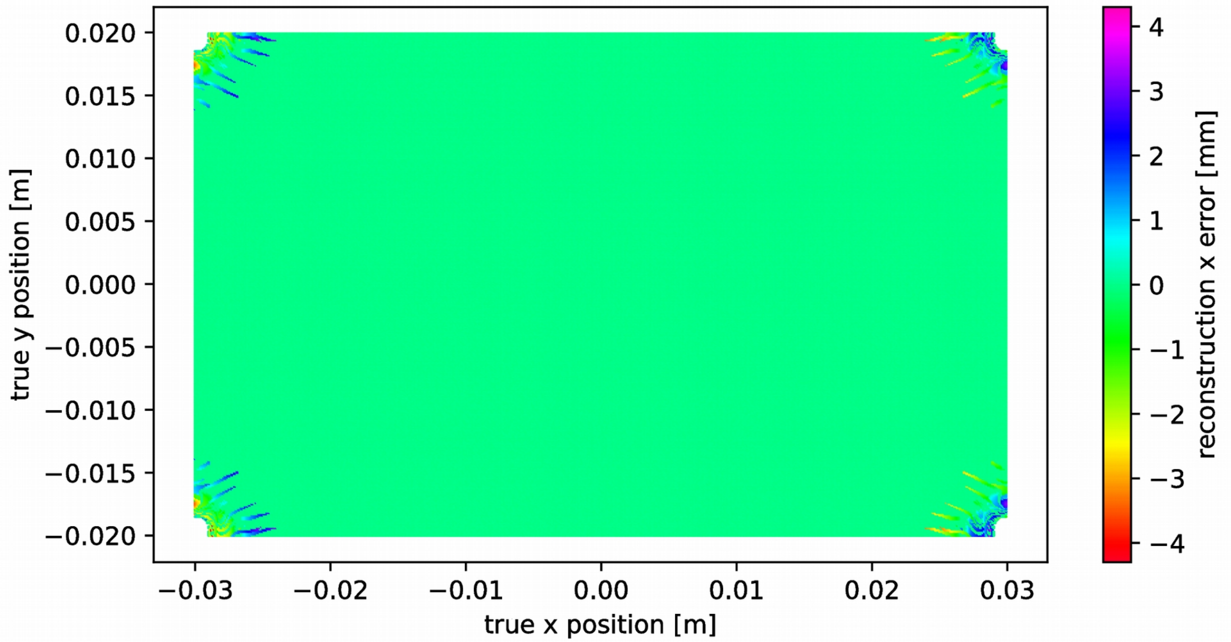


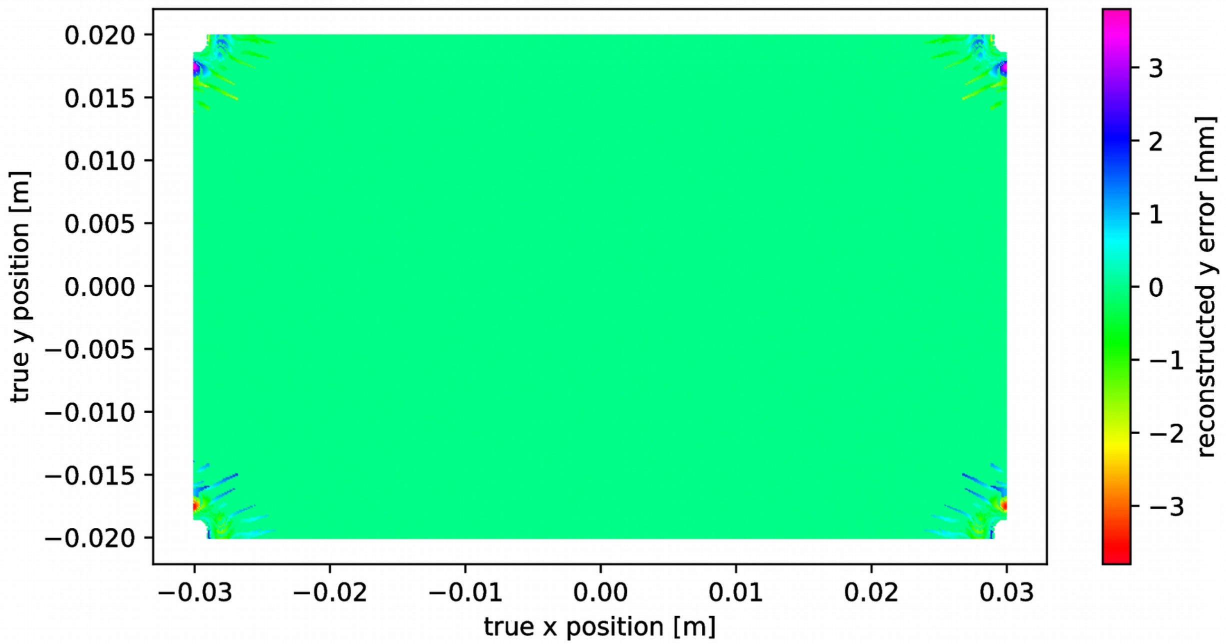


Minimization to extract (x,y)

bpm_arc_xyp.txt

Case 7:

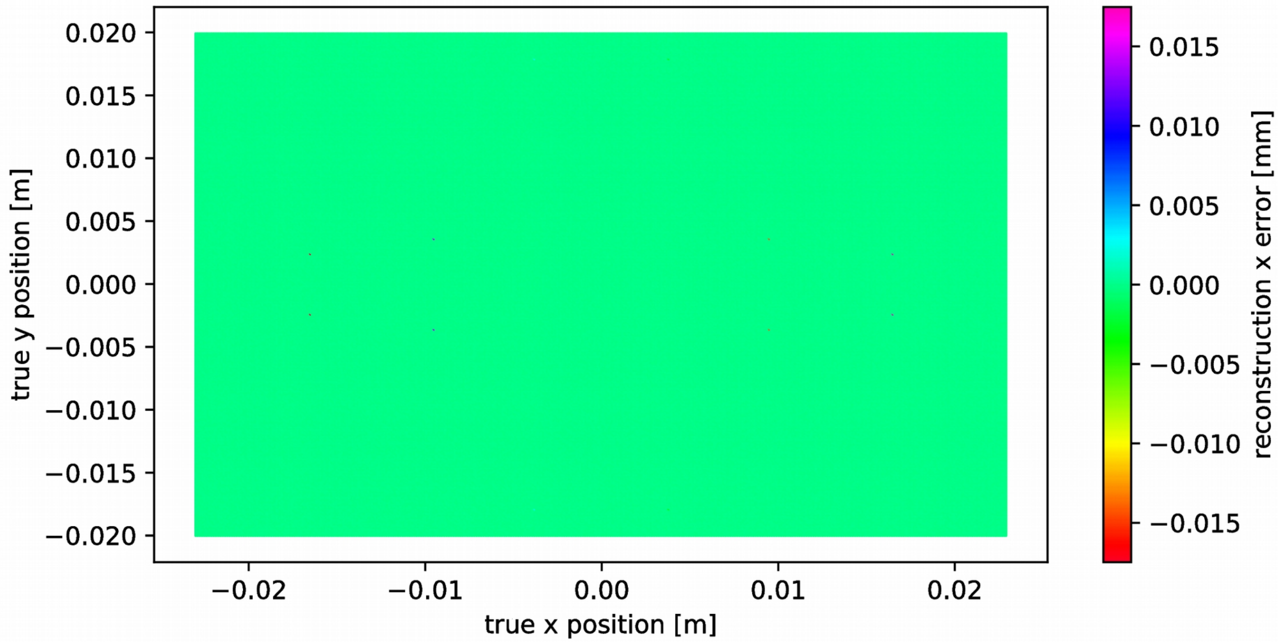


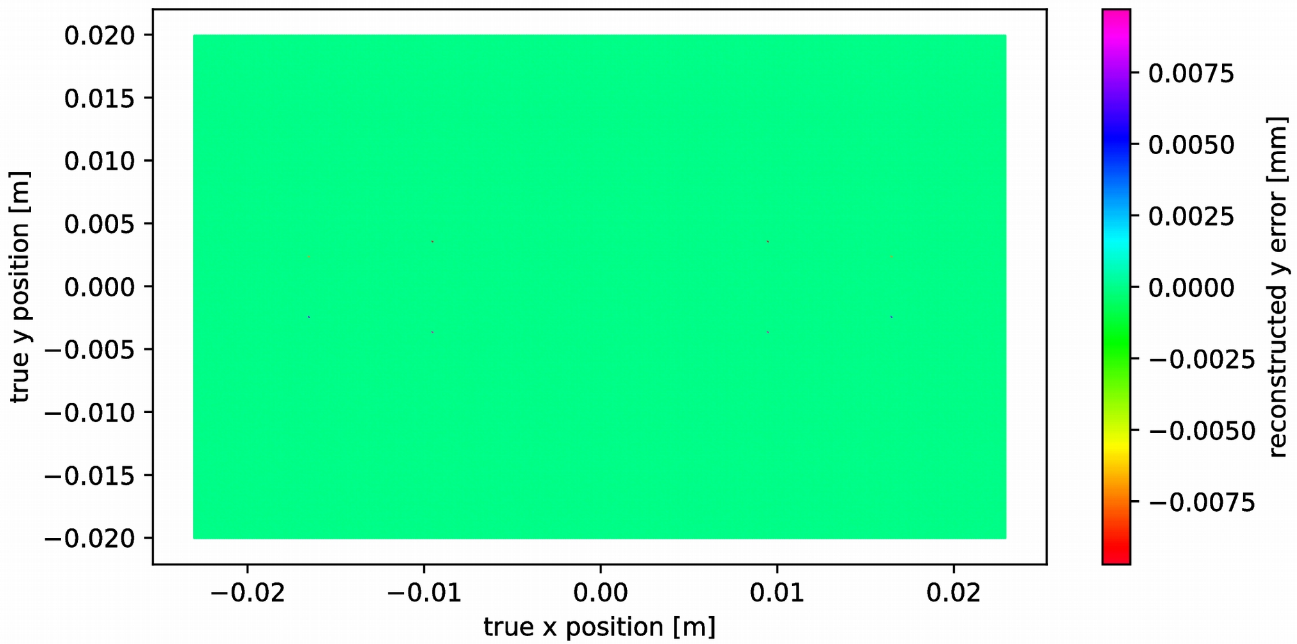


Minimization to extract (x,y)

bpm_arc_xyp.txt

Case 7:





Where do we go from here?

I am not aware of performance/closure test done with CESRV code. Do we know how accurate it is?

What “theoretical” accuracy do we need in reconstructing x and y? In practice, experimental limitations will drive the reconstruction accuracy

Do we want to estimate “experimental” accuracy plugging in experimental errors?

Should we launch the following closure test campaign?

- ✗ draw amplitudes directly from Poisson
- ✗ plug them (with experimental errors?) into CESRV and Python codes
- ✗ compare results

Drawing directly from Poisson will by-pass the closed-loop of using the interpolated look-up table for both generation and reconstruction (which does not test the accuracy of the interpolated look-up table approach)

Additional materials