The JDF module documentation

Piotr Traczykowski

Version 1 : January 22, 2019

1 Introduction

One of Puffin's requirements is that the microparticles are distributed along the electron beam propagation direction (here the z–axis) with sufficient number to allow modelling of the longitudinal electron bunching at the resonant radiation wavelength which gives rise to the FEL's coherent emission [1]. One of the ways to generate the beam is to organise the microparticles into longitudinally distributed 'slices' of given width. The aim of the JDF program is to generate such structure from source data generated by other programs like e.g. Astra or Elegant.

2 Preparing the data

This script is used to convert sparse data distribution to data suitable for Puffin [1]. The principle of this program is to generate randomly placed macroparticles in transverse plane and sliced (ordered) longitudinal structure with desired density.

3 Brief description of the algorithm

The program uses Cumulative Distribution Function (CDF) as operatiing principle. Basing on the source data the three dimensional electron density map is created. Next such map is converted into Probability Density Function (PDF) - i.e. the density is probability of finding electron within selected space. The PDF is converted into continuos three dimensional map using *SciPy* interpolation packages. Next step is to select slice from such 3D PDF and convert it into CDF – actually this is Joint Probality Density Function as we use joint data for x/y plane and the result is also Joint Cumulative Distribution Function. This approach allows to regenerate hollows and gaps within the transverse plane in contrary to method where first x and then y plane is calculated. Final step is to generate random numbers for CDF and then place them on the grid which represents x/y CDF. Last step is to reverse the process and

knowing the CDF value for each randomly generated numbers place it with corresponding x and y position. Such process is repeated for as many slices in longitudinal direction as desired.

4 Usage

The script is easy to use. As it is Python program it requires user to have Python 2.7.x installed together with *NumPy, Tables, SciPy and MatPlotLib* (the program generates some plots during work). Most of the users will only change the amount of particles at the end and number of particle slices per wavelength and FEL parameters like a_u and k_u. But there are more parameters to change. Part of the parameters or rather 'fitting routines' require some deeper knowledge of fitting functions – Python manuals are quite good source of it. All parameters are stored in separate file named *PARAMS_JDF.py*. User is supposed to change the parameters in the file. If the parameters is not found in the file then the default value is used – however, the default values are for a selected FEL design and may led to completely unexpected results.

```
k_u=228.3636
a_u=1.0121809
```

The above are FEL design parameters and should be changed by user depending on target FEL design to be simulated with Puffin.

```
SlicesMultiplyFactor=10
```

This value is setting how many electrons will be generated per each λ_r value in z–axis. The λ_r is calculated and printed on the screen for the given beam at the beginnig of calculation. The script then displays the number of particles in one slice along z–axis. If user feels unsatisfied with the value then one can abort run and change the value.

```
NumOfSliceParticles=1000
```

This parameter defines the number of macroparticles within each slice. The important note here is that user has no impact on each macroparticle weight – the weight is calculated using total initial charge of the actual slice in the beam and the number of particles within this slice.

```
BeamStretchFactor=0.0
```

This is smoothness factor. This parameters tells the program that the size of the beam should be extended along Z–axis both ways (0.015 means that beam start and end values will be extended approximately by this factor). The larger the parameter then the extension is longer and therefore the curve goes more smoothly to zero. Of course setting this

parameter to value different than 0.0 extends the beam and user should be aware that it will add some macroparticles to the beam start and end although the total charge of the beam is kept the same due to charge resacle at the end.

```
X_DensitySampling=80
Y_DensitySampling=80
Z_DensitySampling=80
```

The above sets of parameters is defining initial sampling of the beam. The whole beam electrons density composition is probed using above 'bins' in each direction. In this case we talk about electron not macroparticle density as this is weighted statistics and the weight depends on each macroparticle charge. The larger the number of the '_DensitySampling' the more precise shape of charge curve in z–axis and then x/z–plane and y/z–plane is obtained. However, many users wish to have smoother current curve in z–axis and therefore the low number of bins in z–axis is advised (e.g. 20 or even 10). In case of very complex electron density shapes that user is sure he needs them bin number should be increased.

The last but not least options are user possible changes in fitting functions used to reproduce shape, charge or momentum of the original beam. These functions are set to default values and in most cases work well without any changes. However experienced user might want to modify the functions. Starting from the beginning we have:

```
HxHyHz=ndimage.gaussian_filter(HxHyHz,1.0)
```

The parameter here is used to smoothen the three dimensional electron density map. If user wants to switch it completely then commenting this line is advised. The parameter smoothens the distribution thus making the interpolation function in next step smoother and speeding up the calculations. It also removes some tiny artifacts which possibly occur during the electron density sampling. Value of 1.0 is safe in most cases.

```
#interpolator=interpolate.LinearNDInterpolator(xyzpoints,
    positions[:,3],rescale=True,fill_value=0)
interpolator=interpolate.NearestNDInterpolator(xyzpoints,
    positions[:,3],rescale=True)
```

The user can choose here between two interpolating functions for electron density map. The NearestND is faster but sometimes creates 'sharp' edges while generating particles (therefore we use the Gaussian filter before). The LinearND function creates smoother distributions of particles but is much slower (this can vary but user can expect 10–100x slower calculations). The default option is the NearestND.

```
Full_PX=interpolate.griddata((...) method='nearest',rescale=True
    )
Full_PY=interpolate.griddata((...) method='nearest',rescale=True
    )
Full_PZ=interpolate.griddata((...) method='nearest',rescale=True
    )
```

The momentum interpolation in lines 389–397 works on different principle than particle generation. In general it maps the old momentum data (using source particle positions and momentum) on new particle positions. Here, the user can choose between 'nearest' and 'linear' option. The default is 'nearest' however in some cases using linear may generate better results. The

References

[1] L.T. Campbell and B.W.J. M^{c}Neil, Phys. Plasmas **19**, 093119 (2012)

[2] B.W.J. M^{c}Neil, M.W. Poole and G.R.M. Robb, Physical Review Special Topics – Accelerators and Beams Vol 6, 070701 (2003)