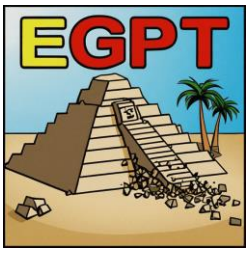
ASTeC-AP-EGPT-01.1 v0.1
Origin: 22/08/2024
Updated: 21/10/2024
joe.crone@stfc.ac.uk

The EGPT Code



Joe Crone
Accelerator Physics Group
Accelerator Science and Technology Centre
STFC Daresbury Laboratory
Warrington WA4 4AD, United Kingdom

Keywords EGPT, GPT, Jitter Analysis, Error Analysis, Code Development, Documentation, RUEDI, EPAC

Summary

EGPT (Errorable General Particle Tracer) is a python-based package which acts as a wrapper for the General Particle Tracer code to simplify jitter and error analysis studies. EGPT takes in a GPT input file, allows the user to set tolerances, run the errored GPT trails and analyse the results. This package aims to be as generalised as possible; however, some conventions must be set. This report will outline how to use EGPT, explain how the code functions and set out the limitations and future developments.

This EGPT technical note will be updated with any additional developments to this code. Major additional developments and supplementary material will be covered in their own technical notes and summarised here.

Contents

Introduction

The EGPT package is a python-based wrapper for performing error and jitter analysis studies on General Particle Tracer (GPT) [1].in lattice files. EGPT takes a user-generated GPT .in file and generates an errored lattice which when passed a set of tolerances, specified in a YAML configuration file, can be used to run errored/jittered GPT runs and provide analysis of the result. EGPT aims to be a generalised package for error and jitter analysis but due to the quirks of the GPT co-ordinate system specification requires a convention for dipoles.

The EGPT package is currently hosted on Gitlab at EGPT and the status of the package is presented best on Gitlab. Currently, EGPT has been applied to energy jitter studies for the RUEDI imaging line and misalignment analysis of the EPAC PMQ capture and focus array. EGPT can be extended and incorporated into more complex studies, for example for replication of accelerator measurements with errors.

EGPT allows implementation of errors on all relevant GPT elements. For example, applying errors to elements used to set the bounds of the simulation or transform the beam and co-ordinate system would be meaningless and these are excluded. Errors can be applied to accelerator elements such as misalignments (including rotation) and parameter errors, for example errors on the field strength of a quadrupole or phase of a cavity.

This technical note sets out the function of the EGPT code and provides a simple example of how to run the code. More in-depth details of the functionality of the code are then provided. A full worked example using the EPAC permanent magnet quadrupole (PMQ) array [2] is given. The limitations and future developments of the EGPT code are then discussed. Knowledge of the GPT code is assumed throughout this technical note with reference to the GPT v3.43 manual [1].

EGPT Overview & Simple Guide

Overview

A GPT input file is supplied and the EGPT code will assign a series of error parameters to the element including misalignments (dx, dy, dz), rotations (θ) and both fractional and additive (f and d) errors to parameters. A template for the YAML

tolerance file is also generated. Users can then implement their mean and tolerance values and chosen model (Gaussian or uniform) to generate the errors from. Assigned errors are randomly generated from this model.

The errored lattice and tolerance file are then supplied to assign and generate the random errors specified and run the errored GPT lattice for a single or multiple trials. This generates a series of temporary GDF beam files, which are of identical form to those produced by GPT and the relevant time-like and position-like GDF analysis files generated using the GDFA program in GPT. As standard the analysis files are kept, and the beam file is discarded because the volume of data from multiple beam GDF files is prohibitive (many GBs).

Analysis of the GPT generated data (GDF files) makes used of the EasyGDF [3] package developed by C. Pierce, which parses GDF files from GPT into a python dictionary. A series of analysis scripts, which analyse the GDFA produced files, and generalised plotting functions are created for easy plotting of the time-like and position-like analysed data.

Simple Guide

Using the provided run file GPT_error_run.py a simple EGPT run can be conducted by several python commands and editing of the generated tolerance file. Here we consider that the user would use the default GPT_error_run script which provides an example of how EGPT can be used for generating, with a template file for tolerances, then running an errored lattice file and analysing the result with trajectory and beam size plotting. The user can modify the GPT_error_run.py script to add in any additional analysis or plotting desired.

The simple workflow of this error simulation would be:

- *(User specifies their GPT path and license number in environment variables, see Startup & Installation)*
- The user adds their GPT lattice (.in) file to a directory of their choosing.
    - The lattice should be modified to comply with the EGPT dipole format (see [cite] for details)
- The user runs a template generation run by running a command from the E_GPT working directory of the package python GPT_error_run.py <path_to_lattice>/<lattice_name>.in
    - Generates an errored lattice <path_to_lattice>/<lattice_name>_ERR.in
    - Generates a YAML template file for tolerances <path_to_lattice>/GPTin_tolerance_temp.yml
- The user then populates the YAML template file with their specified tolerances for each element
- The user then runs an error run by again running a command from the E_GPT working directory python GPT_error_run <path_to_lattice>/<lattice_name>.in <path_to_lattice>/GPTin_tolerance_temp.yml <no._trials>
    - User specifies the number of trials <no._trials> to perform
    - GDFA position and time data is returned to the user in the form <path_to_lattice>/temp_<pos|time>_<trial_no>.gdf

    - Beam data is not returned as standard (the keep_beam = True flag must be set)
    - A file of the specific errors applied is returned error_applied.dat
- The plotting and analysis of the data specified by the user is returned
    - Plots are auto-saved

EGPT Functionality

This section gives a more in-depth overview of how EGPT functions and how to obtain and run EGPT.

Startup & Installation

EGPT is currently hosted on both Gitlab (internal) and Github (open-source). The internal STFC Gitlab version contains several proprietary examples including examples of EGPT applied to both the RUEDI and EPAC projects. Currently the Gitlab iteration is best maintained.

To install EGPT the files need to be pulled from the repository, and the required packages installed. EGPT is currently built for Python 3.10 and up to date for v3.43 of GPT. Effort will be made within future development of EGPT to properly package a stable build.

Environment variables must be configured for GPT for EGPT to function. The GPT license number must be supplied in an environment variable named GPTLICENSE. The path to the GPT executables should also be supplied. This path will typically be “C:\Program Files\General Particle Tracer\bin\”. The path should be added to the PATH environment variable.

Code Structure

Several scripts are used in the construction of EGPT including GPTin_error_modifier.py, GPT_run_analyse.py, GPT_plotting.py and GPT_error_run.py. The GPT_config file must also be included in the distribution, which sets up the path and license for GPT. The current structure is shown in Fig. 2.

GPTin_error_modifier is responsible for creating an errored lattice file and the accompanying tolerance template. GPT_run_analyse applies the tolerances from the tolerance yaml file, runs GPT for the specified number of trials, runs the in-built GPT analysis (GDFA) on the produced data and returns the data in a dictionary format useable by GPT_plotting. GPT_plotting is a set of analysis and plotting scripts which allow the user to plot the generated GDF data in a standardised format and use standardised analysis methods on the data.

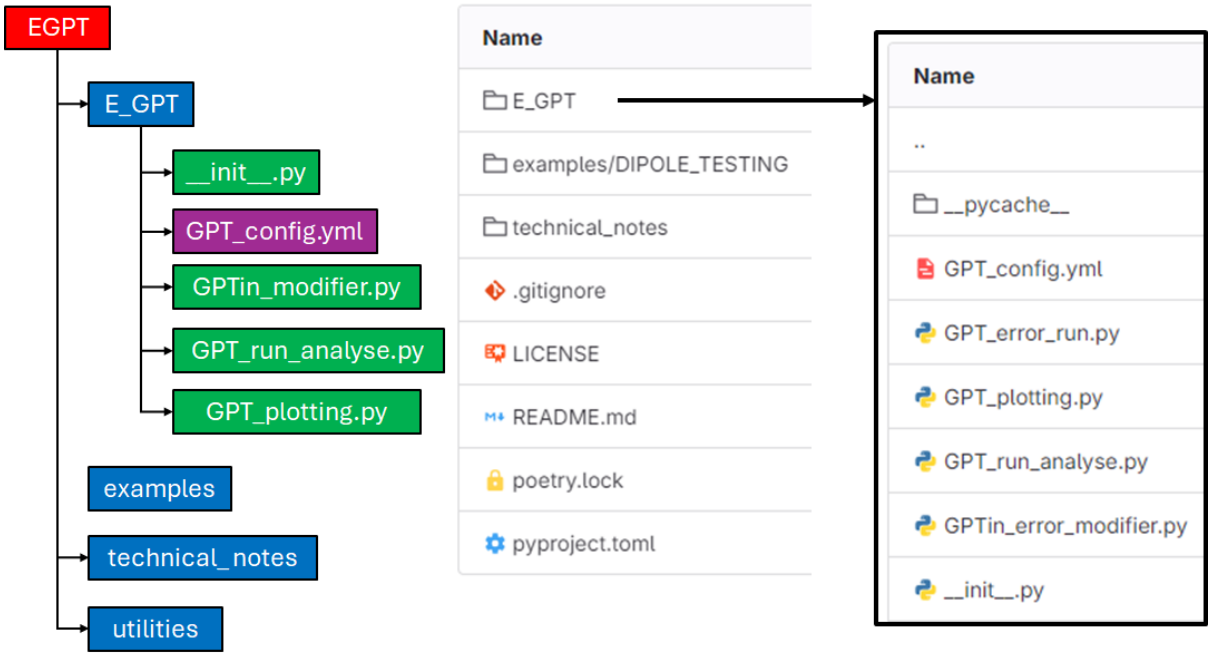


Figure 1. Structure of the EGPT code. Schematic of the package structure and excerpts from the Gitlab repository.

The EGPT code requires many standard packages such as NumPy, munch, PyYaml, re, itertools, subprocess, os, multiprocessing SciPy and Matplotlib. In addition, the EasyGDF package [3] is used for parsing GDF.

Erroring the Lattice File

Errors are applied to each command in the GPT lattice file that is denoted as a beamline element or initial condition. This excludes all GPT commands that change the settings of a file, modify a co-ordinate system or generate output, for example accuracy(),ccs(), ccsflip(), XYmax, spacecharge3Dmesh(), tout() and screen() etc. Erroring these parameters would be meaningless. Currently, initial conditions are not errored, but these shall be in future versions of EGPT.

This section discussed how errors are applied to the GPT lattice file. The applied errors can be split into three types: element co-ordinate system (ECS) errors, parameter errors and initial condition errors. Element co-ordinate system errors change the position and geometry of an element in the beamline and include misalignments and rotations. Parameter errors involve errors applied to the element itself, such as its length and strength for magnetic elements and its phase and amplitude for elements such as RF cavities. Initial conditions errors are concerned with the initial generated bunch, for example errors to the initial transverse size, energy or position of the bunch. An example implementation of an errored quadrupoles magnet is shown in Fig. 2.

```
quadrupole("wcs",0 + dx1,0 + dy1,0.3 + dz1,cos(th1),-sin(th1),0,sin(th1),cos(th1),0,f_11_1*0.2 + d_11_1,f_12_1*50 + d_12_1);
```

Figure 2. Errored quadrupole implementation within the GPT lattice file. This example has been generated from an inputted GPT lattice by EGPT.

Errors are typically applied to the lattice using the lattice_replacer_template() method of the GPT_error_mod class, which generates both an errored lattice and the tolerance template (discussed in the next section). The lattice_replacer() function is primarily responsible for generating and errored lattice. This method iterates through the lattice file line by line and each line determined to have an element is edited by the element_replace() function.

Elements are determined by the element_types() command which compares the first word of a each line to a set of known errorable GPT commands from the manual, a list of which is provided in Appendix 1, which can be shown using the supported_elements() method. This also involves the element_splitter() method which uses the iselement() function to exclude all commented lines and variables and account for small special handling of elements such as collimators.

Misalignments & Rotations

Assigning misalignment and rotations errors to elements means that the element coordinate system (ECS) for the element must be modified. The ECS in GPT can be specified in numerous ways – a full specification or shorthand for positioning longitudinally (and longitudinally with rotation in later GPT versions) and application of the misalignment and rotation errors must account for this. Currently only the roll of the elements about the longitudinal axis can be errored; pitch and yaw of the elements will be added later (see Rotations), following the typical definitions from aviation.

The misalignment terms dx, dy and dz are only considered as additive errors upon the beamline element. The rotation error θ is considered also to be an additive error. Additive errors are added to the original specified element values. For example, an element that is initially rotated (e.g. a skew quadrupole, rotated about the longitudinal axis) can be assigned a rotation error without overwriting the initial rotation specification.

The ECS_replacer() method is responsible for replacing the original element co-ordinate system and producing a full ECS specification with errors. This has special handling for the sectormagnet command, to adapt the dipole element co-ordinate systems.

Erroring of Parameters

Errors are applied to parameters in the form f_<param_no>_<ele_no>*<param_val> + f_<param_no>_<ele_no>, where f and d denotes whether the errors are fractional or additive, <param_no> is the argument number passed to the element command and <ele_no> is the number of the element in the lattice file; these are numbered consecutively based on the elements recognised by EGPT to be beamline elements as shown in Appendix 1.

Parameters of the elements are errored by the param_replacer() method which assigns the fractional f_ and additive d_ errors to each of the parameters whilst leaving the original value unaltered. Parameter errors are only applied to numerical parameters that occur within the element parenthesis beyond the ECS (which has a known number of variables) and string-based parameters within the parenthesis of the element such as fieldmap files are left unaltered. This criteria is used to define

the errorable parameters. Again, special handling is employed for dipole elements as explained in the Dipoles section.

Collimators

Collimator elements such as scatteriris() can be errored in EGPT similarly to any other element using misalignment and the form f_<param_no>_<ele_no>*<param_val> + d_<param_no>_<ele_no> for their parameters. Special handling is used to account for the collimator() scatter = ”scatter_model” form that these are entered in, which differs from all other GPT elements. The scatter model is defined by an element such as forwardscatter(), which is not errored. The scatter model in the collimator() scatter=”scatter_model” cannot be errored or varied.

Dipoles

Handling of dipoles in EGPT is a special case because dipole elements such as isectormagnet() and sectormagnet() change the co-ordinate system used in GPT. Misaligning dipoles in GPT consequentially requires several changes of co-ordinate system. **Note that currently dipole implementation using only the sectormagnet() command is supported**.

Additiona;lly, the parameters of the sectormagnet() command, as shown in the GPT manual [1] in Fig. 1, are typically dependent on each other (R and Bfield) or dependent on a parameter defined elsewhere, for example the bend angle upon which R, Bfield phiin and phiout depend. If different errors were applied to Bfield and R without being propagated correctly then the physics of the element would no longer be consistent which must be avoided.

```
sectormagnet(fromCCS,[finalCCS],toCCS,R,Bfield,[phiin,phiout,dl,b1,b2]);
```

Figure 3. Sectormagnet() command in GPT. Its arguments are dependent on each other and other specified parameters.

This means the parameters of the GPT dipoles as well as the co-ordinate systems must also be handled differently to other elements in EGPT. For EGPT to function correctly, dipoles in the initial GPT file must be setup using the convention as in the GPT manual [1] with numbering. For example, dependent parameters such as bendang (the bending angle) are specified as bendang_<dipole_number> and the bent co-ordinate system specified as bend_<dipole_number>. The parameters of the dipole must all be defined variables; however, these variables can be defined based upon each other.

This convention has been developed to correctly error dipoles. For more details on proper handling of dipole errors and misalignments see the dedicated technical note ASTeC-AP-EGPT-02 [4] which explains explain how to correctly setup dipoles for use in EGPT and the convention in more depth. An example of a correctly configured dipole in the GPT .in lattice file is shown in Fig. 5.

```
bendang_1 = 20;
Ldip_1 = 0.15;
phiin_1 = 0;
phiout_1 = 0;
dl_1 = 0;
b1_1 = 0;
b2_1 = 0;

Rbend_1 = Ldip_1/(bendang_1/deg) ;

Bfield_1 = (Po*1E6)/(Rbend_1*c);

intersect_1=Rbend_1*tan(bendang_1/(2*deg));

ccs("wcs", 0, 0, pre_compressor_distance+intersect_1, cos(bendang_1/deg), 0, -sin(bendang_1/deg), 0, 1, 0, "bend_1");

sectormagnet("wcs", "bend_1", Rbend_1, Bfield_1, phiin_1, phiout_1, dl_1, b1_1, b2_1);
```

Figure 4. Example of a sectormagnet() dipole setup correctly for the EGPT convention. All dipole parameters must be pre-defined by calculation from other variables or assignment. The bend co-ordinate system and the sectormagnet() command must be specified as shown.

The sectormagnet() command itself is errored similarly to other elements.The ECS replacer() has special handling to correctly rename the dipole entrance and exit co-ordinate systems. The parameters of the sectormagnet command are modified to become their errored versions for example, from bendang_1 to bendang_err_1, where the errored parameters are generated by a special handling add_dipole_err_params() method.

The sectormagnet dipoles use two additional functions: add_dipole_ccs() for misaligned co-ordinate system generation and add_dipole_err_params() for generation of the errored dipole parameters and consistency between these. These functions are used after the rest of the lattice has been modified in the lattice_replacer() method.

The add_dipole_ccs() command generates the entrance and exit co-ordinate systems of the dipoles, which are misaligned, using the ccs() GPT function. There is also two ccsflip() elements generated, which change between the nominal and misaligned co-ordinate systems at the entrance and exit of the dipole. Further details on this process are available in ASTeC-AP-EGPT-02 [4].

The add_dipole_err_params() generates the errored variables for the sectormagnet command. These are errored in the form f_<param_no>_<ele_no>*<variable> + d_<param_no>_<ele_no>, similarly to values of straightforward elements. Here though the <param_val> has been replaced with the <variable>, where an equation based on other variables that can be errored (such as Rbend_1 in Fig. 5) can be substituted into this form. This means that parameters of the sectormagnet can be errored consistently. Further details on this are provided in ASTeC-AP-EGPT-02 [4].

Setting Tolerances & Applied Errors

Explanation of how the code works is required

The Tolerance File

EGPT produces a tolerance template which has a form shown in Fig. 6. Here the YAML file has a nested structure where the top level is <ele_name>_<ele_no>, then

there is a list of errors that can be applied such as dx, dy, dz misalignments, θ rotations and parameter errors f_ and d_.

```
quadrupole_1: # <- errored element name (<ele_name>_<ele_no>)
  dx1: # <- errored parameter (<param_identifier><ele_no>)
  - 0 # <- mean error value
  - 0 # <- tolerance value
  - gaussian # <- error distribution model
  - 3 # <- cut-off value
```

Figure 5.Standard form of the generated tolerance template. Users specify their tolerances within this template.

Two error models – Gaussian or uniformly distributed errors are currently available to users. Varying the mean error value can be used to create a skewed distribution the error is sampled from, for example if a magnet can't achieve its design field. The tolerance is the width of the sampling distribution. The cut-off defines the endpoint of the sampling distribution which has limits ±<cut-off>*<tolerance>.

Parameters that are left unaltered within the template will result in no error being applied to that parameter. If EGPT is ran with an unaltered tolerance template, then this is equivalent to running the unaltered lattice.

The tolerance template is generated using the lattice_replacer_template() method of the GPT_error_mod class. This takes the list of the errored parameter names and numbers and the errored element names and numbers produced by the element_types() and parameter_name_sorter() to generate a nested munch dictionary. Parameters are identified as explained in the above section.

The much dictionary is structured with the <element_name> then the errors for this element which are denoted for misalignments by the form <parameter_name><element_#>, or for errorable parameters denoted by the form <f|d>_<parameter_#>_<element_#>. Each of these errorable parameters has a sub-dictionary which consists of the mean value, tolerance, distribution type and cut-off for sampling that error, as explained in the next section. The munch dictionary is then converted into a YAML file, which is the This hierarchy of the template is shown in the example in Fig. 6.

Default values of the mean value, tolerance, distribution type and cut-off are set when the template is produced. Here we select a mean value of 1 for fractional and 0 for additive errors as well as a tolerance of 0 because this means running the tolerance template will just replicate the standard lattice. The distribution type and cut-off are set to gaussian and 3 because these are the most used.

Running Error Analysis with EGPT

This needs more information, how does the code work?

EGPT allows the user to define either a Gaussian or uniform error distribution (model) whilst supplying a mean value, a tolerance and a cut-off parameter. These are used to generate an error distribution, where the tolerance is the standard deviation with a cut-off of ±<cut_off>*<tolerance>, from which error values are randomly sampled.

The assigned errors are unique for every element and for each run. For fractional errors the default mean value is typically 1 whereas for additive values it is zero. Setting a mean error that varies from this value is useful in adding a systematic component to the error for example, if a magnet is known to have some systematic misalignment from survey or if a magnet can't meet the required field specification. Cut-offs on the error distribution prove useful in removing highly unlikely cases from the simulation.

Errored GPT runs are computed in parallel in EGPT using the pool.apply_async() function from the multiprocessing package. This improves the speed of running multi-trial error analysis using EGPT, especially when using a cluster. Parallel computation is required because GPT runs will need to consider space charge or high-accuracy computation (for example in the low-energy RUEDI accelerator this code was designed for).

EGPT typically discards beam data from each individual error trial as multiple GBs of data are typically generated for ~10 trials of a short < 10 m beamline. Instead, the GDFA analysed data is kept and the beam data is discarded. However, some user cases may require all the beam data and a keep_beam flag is included (keep_beam=True) specified for the run in the GPT_run_get_analysis_dict() method of the GPT_error_run.py script.

Analysis

This needs more information, how does the code work?

EGPT contains a series of analysis and plotting scripts within the GPT_plotting class. This takes in the EGPT data in the munch [4] dictionary format returned by the get_analysis_dict() methods of the GPT_run_analyse class, as explained below. This class is specifically designed for users to add additional analysis and plotting routines, the code in this class will not be exhaustive and should be extended collaboratively. Here this aims to provide a generalised approach to data handling, analysis and data plotting.

Running EGPT with the GDFA analysis - GPT_run_analyse.GPT_run_get_analysis_dict() for error running and analysis or GPT_run_analyse.get_analysis_dict() for analysis of previously generated data - results in the GDFA data file being read into EGPT as a munch dictionary. The GDFA data file returned in EGPT includes all possible data sets for the position and time analysis, for details of the returned datasets see the GDFA documentation in the GPT manual [1].

The GDF analysis (and beam) file data is parsed using the EasyGDF [3] package and is split into nested dictionaries. The first layer denotes the trial number, with key trial_1, then the sub-dictionaries (second layer) separate the time and position output (time and pos are used as keys), there is then a third layer (a sub-sub-directory) which contains the data arrays for each parameter, such as position or avgx, which is used as the key. Additionally, when the beam data is kept (keep_beam = True) there are keys for touts and screens at the second layer (along with time and pos) which contain the time-like (touts) and position-like (screens) beam data. This is shown in the data hierarchy/anatomy in Fig. 4.

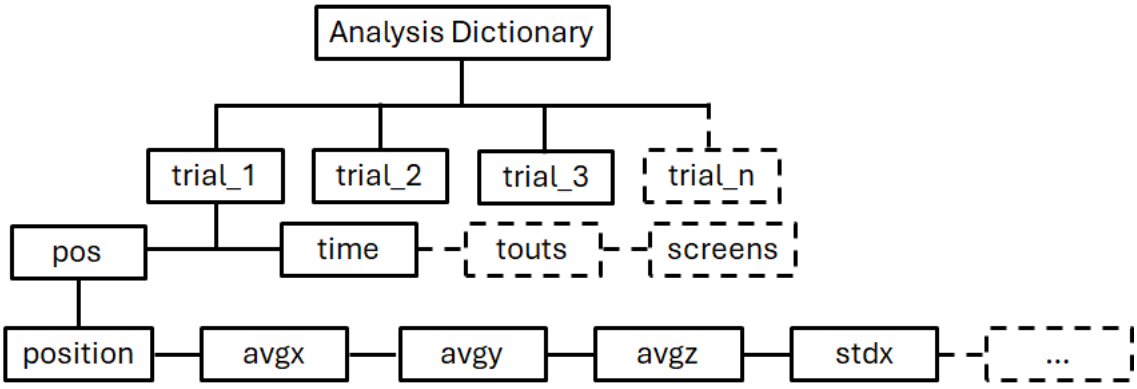


Figure 6. Data hierarchy for the Munch dictionary of EGPT data returned from a multi-trial error analysis run. Many more datasets are available from the GDFA program in GPT, see the GDFA documentation for details.

Position Plotting & Co-ordinate Systems

The order of the GPT positional output can be confusing once changes of co-ordinate system have been introduced. For example, screen output in co-ordinate systems post-dipole begins with a position value of 0 and the order of analysed output depends on the listing of the screen commands in the GPT input (.in) file. Therefore, the positional data must be parsed and resorted.

Positional data is parsed and ordered based on the avgt parameter, which denotes the mean time elapsed with respect to the macroparticle beam from the start of the simulation. Therefore, these values are sorted into chronological order and the indexes of the data noted. This indexing can then be used to generate proper ordering of the parameter data, such as the avgx or stdx values.

Analysis & Plotting

Plotting functions currently constructed involve generalised plots of the beam size and the trajectory of the beam as well as tools for energy jitter analysis. These tools are still under development and can be fully generalised, however these are of low priority because users should be able to easily extend the analysis and plotting functions, and the package can never be fully comprehensive for all uses.

The generalised beam size and trajectory plots can be constructed with the time or position analysis data using TP_flag= ‘pos’|’time’ and plot this data for each different trial as a data set in the plot. This plotting will be made more general, allowing the user to specify the data type to plot (there is no need for two functions for beam size and trajectory plotting).

Other energy jitter analysis provides statistical standard deviation analysis of the central energy and energy spread variation of many error trials. Eventually, like the above plotting, this will be made into a generalised function.

Example: PMQ Channel

Limitations and Future Developments

EGPT currently only supports GPT up to version 3.43, however, since all versions of GPT are backwards compatible, future versions are expected to be widely compatible with EGPT. New features, such as the new element rotation system, are currently not supported but will be in updates to EGPT. EGPT will be updated to the latest stable build of GPT.

Initial Conditions & Beams

Handling of initial conditions comes in two forms: handling of bunches generated in GPT (GPT generated) and handling of imported bunches from file (file generated). If a setparticles() command is detected in the lattice file then the bunch is assumed to be GPT generated.

When the bunch is GPT generated initial conditions commands used to generate bunches with GPT such as setxdist(), setGdist() and addxdiv() are errored similarly to parameters. The parameters of these that do not represent the set or distribution type are errored in the form f_<param_no>_<initial_condition_no>*<initial_condition_param> + f_<param_no>_<initial_condition_no> where initial_condition_no is the number of previously initial condition commands specified in the lattice. These initial condition parameters will then appear in an initial_condition section (equivalent to the element name) of the tolerance YAML file. The errors on these parameters will be definable identically to elements of the lattice file. These errors will be read in and applied as a special case.

Alternatively, if the bunch is imported from file using a setFile() command then the tolerance file will include a directory_name names will be loaded from a passed directory in the tolerance file and a random distribution will be chosen from those in the directory for each trial.

Fieldmaps

Similarly to the handling of the errored bunch distributions. The errored fieldmaps should be placed into a directory and the directory name passed to the tolerance file. A random fieldmap will be selected for the element on each trial from one of the fieldmaps in the directory. The directory will be set to None as default, and if this is not altered only the existing fieldmap, with errors placed on the element using the usual f_*<value> + d_ form to values such as the strength, will be applied. This modification will be made so that the traditional f_ and d_ type errors can be combined with the fieldmap errors.

Rotations

Only roll – rotation about the longitudinal axis - of the element is currently supported. Pitch (rotation about the horizontal x axis) and yaw (rotation about the vertical y axis) of the elements should also be supported. A diagram of the co-ordinate system of an element with these rotations is shown in Fig.

This involves modifying the element co-ordinate system defaults to contain the full 3D rotation matrix for roll, pitch and yaw as given by

Where … There is then a series of angles θ_x (pitch), θ_y (yaw), θ_z (roll) defining the position of the element. All these angles can then be errored.

Analysis

References

[1] S. B. van der Geer and M. J. de Loos, “General Particle Tracer - User Manual (Version 3.43),” Pulsar Physics, Eindhoven, 2023.

[2] B. Muratori et al, The EPAC Electron Transport Beamline - Physics Considerations and Design, Venice, Italy: IPAC 2023 - TUPA 105, 2023.

[3] C. Pierce, EasyGDF, https://github.com/electronsandstuff/easygdf, 2020.

[4] D. Sternlicht, munch, https://github.com/Infinidat/munch, 2023.

Appendix 1: List of GPT Elements Errorable in EGPT

Any element that has an ECS is likely errorable. Only elements without an ECS or those that control the overall boundaries of the simulation are un-errorable. Currently initial conditions elements are neglected from this list. The llist of errorable elements in EGPT is shown in Table 1.

Table 1. Errorable elements in EGPT

TE011cylcavity	TE011gauss	TE110gauss	TErectcavity
TM010cylcavity	TM110gauss	TM110cylcavity	TM010gauss
TMrectcavity	trwcell	trwlinac	trwlinbm
circlecharge	Ecyl	ehole	erect
linecharge	platecharge	pointcharge	barmagnet
Bmultipole	bzsolenoid	linecurrent	magline
magplane	magdipole	magpoint	quadrupole
rectcoil	sectormagnet	sextupole	solenoid
map1D_B	map1D_E	map1D_TM	map2D_B
map2D_E	map2D_Et	map2Dr_E	map2D_V
map25D_E	map25D_B	map25D_TM	map3D_E
map3D_TM	map3D_Ecomplex	map3D_Hcomplex	map3D_V
map3D_B	map3D_remove	scatterbitmap	scattercone

scatteriris	scatterpipe	scatterplate	scattersphere
scattertorus	multislit	drift	gauss00mf
undueqfo	unduplan	wakefield	

Notably rectmagnet and isectormagnet are not yet accounted for because, like sectormagnet, these require special handling as they are dipole magnets that vary co-ordinate systems. Erroring these is more difficult.