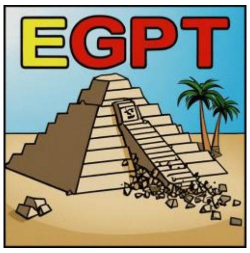
ASTeC-AP-EGPT-01.1 v0.1

22/08/2024

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, Jitter Analysis, Error Analysis, Documentation, RUEDI

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.

Contents

Introduction

The EGPT package is a python-based wrapper for performing error and jitter analysis studies on GPT .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.

EGPT Overview & Simple Guide

Overview

A simplified description of how EGPT works

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 [cite] 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

**A simplified description of how to run EGPT **

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.

EGPT Functionality

Detailed explanation of how EGPT works

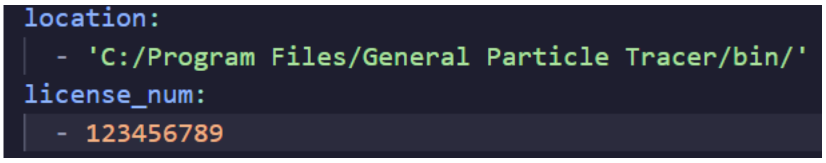
Code examples & things from GPT manual

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. A GPT_config.yml file must be created which contains the GPT location and the license number as shown in Fig. 1.



```
location:
  - 'C:/Program Files/General Particle Tracer/bin/'
license_num:
  - 123456789
```

Figure 1. Example GPT_config.yml file.

Effort will be made within future development of EGPT to properly package a stable build.

Code Structure

The files involved in running EGPT, dependencies & paths

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 structuew is shown in Fig. ?

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 for the GPT output meeting.

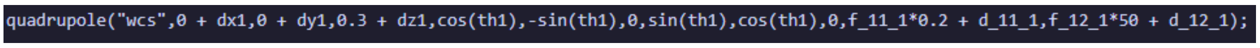
Diagram of the code structure

The EGPT code requires many standard packages such as NumPy, munch, PyYaml, re, itertools, subprocess, os, multiprocessing SciPy and Matplotlib. In addition, the easgdf package by C. Pierce 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 the 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.

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.

Form of the ECS's + how they are modified

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.

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 (excludes initial setup and setting commands).

Initial Conditions

Currently erroring of initial conditions is not supported. The following information outlines their implementation structure.

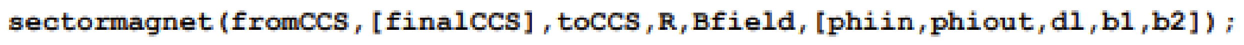
Handling of initial conditions comes in two forms: handling of bunches generated in GPT (GPT generated) and handling of imported bunches from file (file generated). If a setparticles() command is detected in the lattice file then the bunch is assumed to be GPT generated.

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.

Alternatively, if the bunch is imported from file using a setFile() command then…don't know yet!

Dipoles

Handling of dipoles in EGPT is a special case because dipole elements such as isectormagnet() and sectormagnet() in GPT 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**. The parameters of the sectormagnet() command, as shown in the GPT manual 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 long be consistent which must be avoided



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

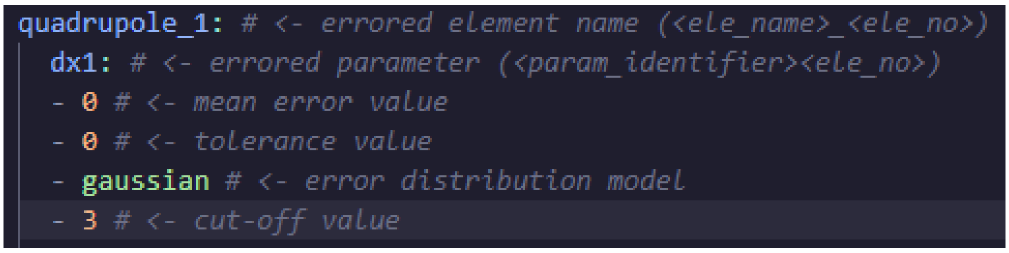
Figure 3. Sectormagnet() command in GPT. Its arguments are dependent on each other and other specified parameters.

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 as specified in the GPT manual [cite], with dependent parameters such as bendang specified as bendang_<dipole_number> and the bent co-ordinate system specified and bend_<dipole_number>. For more details see the dedicated technical note ASTeC-AP-EGPT-02 [cite].

Setting Tolerances & Applied Errors

The Tolerance File

EGPT produces a tolerance template which has a form shown in Fig. 2. 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 4.Standard form of the generated tolerance template. Users specify their tolerances within this template.

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.

Generation of Errors

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.

Running Error Analysis with EGPT

Scripts in EGPT and what they do

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).

Throwing away of data + keep_beam flag

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 …. **Where specified?**

Analysis

EGPT contains a series of analysis and plotting scripts within the GPT_plotting class. This takes in the EGPT data in the standard Munch [cite] dictionary format returned by the get_analysis() 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() for error running and analysis or GPT_run_analyse.get_analysis_only() for analysis of previously generated data - results in the GDFA data file being read into EGPT as a Munch [cite] 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 [cite].

The GDF analysis (and beam) file data is parsed using the easygdf [cite] 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.

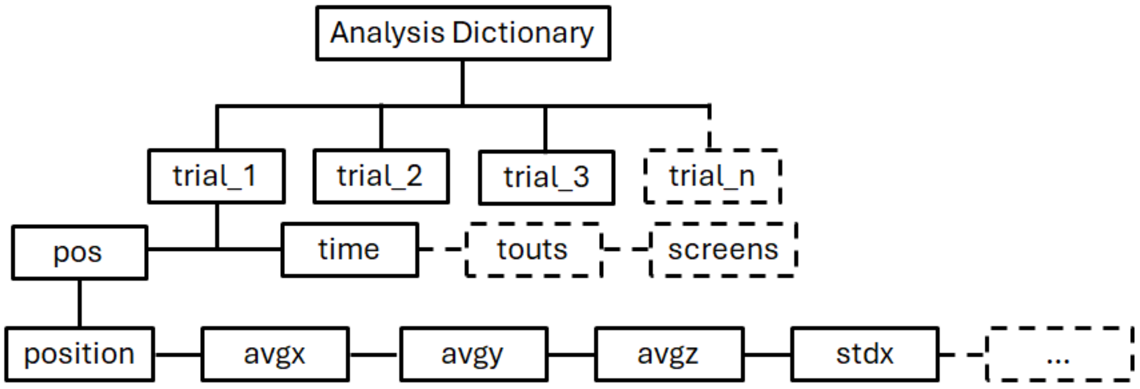


Figure 5. Data hierarchy for the Munch dictionary of EGPT data returned from a multi-trial error analysis run. Many more datasets are available from the GDFA program in GPT, see the GDFA documentation for details.

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

Fieldmaps

Rotations

Analysis

References

There are no sources in the current document.