BBU: Beam Breakup Instability Simulation in Bmad

David Sagan, dcs16@cornell.edu
William Lou, wl528@cornell.edu
August 12, 2023

1 Overview

bbu is a program in `Bmad` which simulates the beam breakup instability (BBU) [1]. BBU occurs in recirculating accelerators due to interaction between the beam bunches and the Higher Order Modes (HOMs) in the accelerating cavities. When beam bunches go through a cavity, they are kicked by the HOM wakefields, and the kick generates orbit distortion. When the bunches return to the same cavity, their off-axis orbit through the cavity create additional wake fields. If the HOM voltage is not properly damped, this positive feedback can lead to instabilities. BBU is therefore a primary limiting factor of the maximum achievable current, the threshold current (I_{th}) in an Energy Recovery Linac. The point of BBU simulation is compute the I_{th}.

2 Simulation detail

The *bbu* consists of two main parts: the core part written in `Fortran`, and the shell part written in `Python`. The user usually sets both BBU `Fortran` and `Python` parameters in the `Python` shell to run the simulations. The user rarely modifies the `Fortran` core.

2.1 Fortran core

The core part determines the stability of a single test current by direct simulation. A train of bunches is tracked though a lattice whose cavity elements can contain HOMs (long-range wakefields). In the program, time (t) is measured in "turns" (abbreviated T). One `turn` is the time it takes a bunch to travel from the beginning of the lattice to the end. At the start of a simulation, $t = 0$, and the HOM voltages in the cavities are set to zero. Bunches are then started at the beginning of the lattice and tracked through to the end. To minimize computation time, a single particle is used to represent each bunch.

Bunches that are initialized in the first turn period, with $0 < t < 1T$, are given a random transverse offset. The offset distribution is Gaussian with default $\sigma = 10$nm . All bunches initialized after the first turn period will have zero transverse offset. In the tenth turn period ($9T < t < 10T$), the “averaged maximum HOM voltage”, $V_{max}(10)$, which is the average of the strongest HOM in all the cavities within this turn, is taken as a baseline to determine whether the voltages are growing or decaying in longer turns. The reason we don’t choose one of the first few turns as the baseline is because HOM voltage variation can be unstable right after initial population of the bunches. Of course, the stability of the test current should be physically independent of the choice. Also, the test current is unstable as long as one HOM voltage is growing. Therefore numerically, we only need to keep track of the strongest HOM voltage, instead of all HOM voltages.

Simulation ends when time hits the nth turn ($t = nT$), in which n is a integer parameter set by the user ($n > 10$ required). The current is declared stable or unstable depending on whether the ratio $V_{max}(n)/V_{max}(10)$ is less than or greater than 1 (or a number slightly above 1, say 1.01, to account for numerical noise), where n is the number of turns simulated (must be > 10) set by the parameter `bbu_param%simulation_turns_max`. (Strictly speaking, the `Fortran` core only outputs the ratio, and the `Python` shell determines the stability solely using this value.) In order to shorten simulation time, ”`limit_factor`” and “hybridization” are implemented, and both are described below.

The main input file for the `Fortran` core is “`bbu.init`”, which looks like:

```
&bbu_params
bbu_param%lat_file_name = ‘erl.lat’  ! Bmad Lattice file name.
bbu_param%lat2_filename = ‘lat2.lat’ ! For DR-scan only.
bbu_param%simulation_turns_max = 50  ! Maximum simulation turn n
bbu_param%bunch_freq = 1.3e9       ! Injector bunch frequency (Hz).
bbu_param%limit_factor = 3         ! Unstable limit to abort simulation.
bbu_param%hybridize = .true.       ! Combine non-HOM elements?
bbu_param%keep_all_lcavities = F   ! Keep all lcavities when hybridizing?
bbu_param%current = 0.1            ! Injected test current value (A).
bbu_param%ran_seed = 100           ! Set specific seed, 0 uses system clock.
bbu_param%ran_gauss_sigma_cut = 3  ! Limit ran_gauss values within Nσ
bbu_param%normalize_z_to_rf = F    ! Shift z to be in range [0, rf_wavelength]?
bbu_param%current_vary%variation_on = F !  Ramp bunch charges up and down?
!bbu_param%current_vary%...              ! Ramping parameters.
/
```

Fortran namelist input is used. The namelist begins on the line starting with "`&bbu_params`" and ends with the line containing the slash “/”. Anything outside of this is ignored. Within the namelist, anything after an exclamation mark “!” is ignored, including the exclamation mark.

Typically `bbu.init` specifies the essential BBU parameters, but not all of them. All BBU parameters (and their default values) are defined in ‘`bsim/code/bbu_track_mod.f90`’. If the user includes extra parameters in `bbu.init`, the user-defined values will overwrite the default values. The main BBU `Fortran` parameters are described below, in alphabetical order:

bunch_by_bunch_info_file (string)

If not empty, the BBU program will output orbit information of all bunches at all tracking stages. The file can be large, and is usually for debugging.

bunch_freq (float)
The injector bunch-to-bunch frequency in Hz.

current (float)
The injected test current in Ampere. With bunch frequency specified, this number equivalently specifies the charge of the bunches ($Q = I/f$). Note that if the bunch charge is `ramped` (See `current_vary%variation_on`), this value is NOT the actual time-averaged current. Instead, this value determines the "reference charge" ($Q_{ref} = I/f$). The user must compute the actual current based on the ramping scheme.

current_vary%variation_on (boolean)
If set to True, the injector bunch charges will be ramped up and down linearly (trapezoids in time). The ramping parameters (with `current_vary` in their name) define the ramping scheme, including `t_ramp_start`, `charge_top`, `charge_bottom`, `dt_plateau`, `ramps_period`, and `dt_ramp` (See Fig. 1).

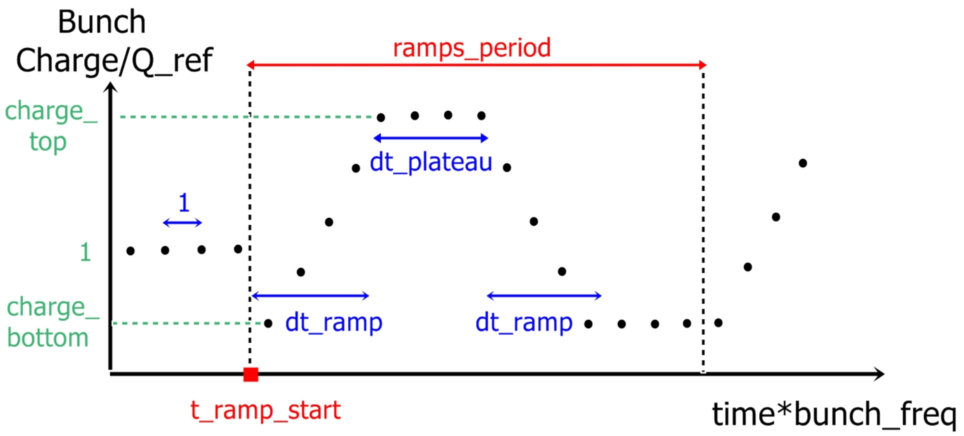

Figure 1: With `current_vary%variation_on` set to True, the bunch charge will be ramped up and down based on the ramping parameters.

Note that all the ramping parameters are dimensionless factors which have to be non-negative. That is, the two charge parameters are in unit of Q_{ref} (See `bbu_param%current`), and all the time parameters are in unit of 1/`bunch_freq`. If both `charge_top` and `charge_bottom` are set to zero, there is essentially no beam. It's acceptable to set `charge_bottom` $>$ `charge_top` to ramp down first then up. Also, it's required that (`dt_plateau` + 2$\times$`dt_ramp`) $<$ `ramps_period`.

drscan (boolean)
Obsolete. The dr-scan functionality has been moved to Python shell. The associated Fortran parameters (including `elname`, `begdr`, `enddr`, and `nstep`) are also obsolete. See the section below on DR-SCAN mode for more details.

ele_track_end (string)
If specified, this is the final element in the lattice the bunches will be tracked to. The associated Fortran variable `ix_ele_track_end` is for internal use and should not be set by user.

hom_order_cutoff (integer)
If positive, ignore HOMs with order greater than this value. Default is zero.

hybridize (boolean)
Hybridization is a process of combining all the (non-cavity) elements between any two cavities into one hybridized Taylor element. The default tracking through a hybridized element is a linear transport (see **use_taylor_for_hybrids**). This can significantly speed up calculation, with possible loss of accuracy.

init_particle_offset (float)
This is the 1σ half width, in meters, of the distribution of the initial transverse coordinates for bunches initialized during the first turn period. The default value is 1E-10, or 10 nm.

keep_all_cavities (boolean)
Relevant only if **hybridize** is set to True. When hybridizing, if this is set to False, then lcavity elements that do not have HOM elements will also be hybridized. This will further speed up the simulation but can lead to inaccurate results. The recommended setting is True (see **normalize_z_to_rf**).

keep_overlays_and_groups (boolean)
Relevant only if **hybridize** is set to True.

lat_file_name (string)
The lattice file (in Bmad standard format [2]) to be used for BBU simulation. The lattice must include at least one multipass line with at least one lcavity assigned with at least one HOM. See Bmad manual for details on lattice specification.

limit_factor (float)
This number has to be greater than 2. For any intermediate turn m $(10 < m < n)$, if the ratio $V_{max}(m)/V_{max}(10)$ is above `limit_factor`, the test current is declared unstable. If $V_{max}(m)/V_{max}(10)$ is below 1/`limit_factor`, the test current is declared stable. In either case, the program aborts so shorten simulation time.

lat2_filename (float)
This file is used for DR-scan only. It specifies the varied length for the 'arc' element in the lattice. See DR-scan mode in the `Python` shell section below.

normalize_z_to_rf (boolean)
Normally, with *normalize_z_to_rf* set to False (the default), the starting phase space z of a particle will reflect the time the bunch that the particle is in is launched. If RF elements are to be hybridized, the hybrid transfer map will not work with the large z values. If *normalize_z_to_rf* is set to True, the starting z values will be set to be in the range $[0, \lambda_{rf}]$ where λ_{rf} is the longest RF wavelength of all the cavities. In this case all the cavities must have commensurate wavelengths.

nrep (int)
Obsolete.

ran_seed (integer)
Random number seed. If set to 0, the system clock will be used, and the output results will vary from run to run.

ran_gauss_sigma_cut (float)
Any randomized value in the lattice, such as initial orbit offset, is limited to a maximum deviation of `ran_gauss_sigma_cut` rms deviations. This is to prevent extreme values from being chosen, which can be un-physical. A typical value is between 3 to 5. The default value is -1, which means no cutoff.

regression (boolean)
Obsolete. The BBU regression test has been moved to be with other regression tests under the bmad-ecosystem.

simulation_turns_max (integer)
The maximum number of simulation turn the program runs up to when no particle loss is detected. Must be greater than 10. A large number slows down the computation, but increases accuracy. A typical choice is between 50 to 300.

stable_orbit_anal (boolean)
If True, write stable_orbit.out and hom_voltage.out (for debugging).

use_taylor_for_hybrids (boolean)
Relevant only if **hybridize** is set to True. Use taylor map for hybrids when True, otherwise the tracking method is linear. Default is False.

write_digested_hybrid_lat (boolean)
For debugging purposes.

write_voltage_vs_time_dat (boolean)
For debugging purposes. If true, write a (large) output file which contains the maximum_-HOM_voltage data at every tracking step. The user can plot the data to visually check the stability of the test current. This is probably the most reliable way to check whether the test current is stable, because the stability behavior can be unclear if total simulation time is too short, or if the no clear divergence or steady-state occurs to the program.

2.2 Python Shell/Wrapper

Since the `Fortran` core program only determines the stability of ONE test current, multiple runs of the core (with different test currents) are required to pin down the I_{th}. This can be done with the developed Python shell which interacts calls the core program. Besides finding the I_{th} of a specific design, the shell can also generate statistics of I_{th} by introducing small variations to either the lattice or the HOM assignments. Each type of variation is considered as a “simulation mode” (See Section below).

The main file for the Python shell is test_run.py, in which the user specifies all the BBU (Fortran) parameters and additional Python parameters. Other associated Python codes are under the $DIST_BASE_DIR/bsim/bbu/python/ directory. The user should make sure this directory is included in the environment variable PYTHONPATH.

The most important Python parameters which are common to all the simulation modes are:

exec_path
The exact path of the compiled BBU Fortran program. Typically this is under the .../production/bin/ directory.

threshold_start_curr
Initial guess of the test current. An educational guess based on the lattice design and HOM assignments can reduce the total simulation time.

final_rel_tol
Python shell iteratively calls the Fortran core to pin down the I_{th}. This value determines accuracy of the I_{th} found.

The existing Python shell has limited functionalities, so users are encouraged to write their own wrapper with external programs for independent BBU studies. For instance, multiple BBU runs with different test currents can be run in parallel. Also, by importing data from the write_voltage_vs_time_dat output, one can write their own function to determine the BBU stability.

3 Simulation Modes

The user can run different simulation modes by adjust the arguments passed to test_run.py. The details of each mode and their additional Python parameters are described below. Note that regardless of the mode, original lattice MUST have a valid initial HOM assignment.

3.1 DR-SCAN mode

Number of argument: 0

Run this command under $DIST_BASE_DIR/bsim/bbu/examples :

```
python $DIST_BASE_DIR/bsim/bbu/test_run.py
```

Obtain (I_{th} v.s t_r/t_b) for a lattice with a varying arc length. (t_r is the recirculation time, and t_b is 1/bunch_freq). The lattice must have a unique element named "arc" of which the length ($t_r\times$ speed of light) is varied. This mode produced the plot for Fig.3 in [2] and Fig.14 in [3], for which one dipole HOM is assigned to one cavity in a lattice with one recirculation pass (the simplest BBU model). This mode is commonly used to check if the simulation agrees with the theory, as shown in the two papers.

`Python` Parameters:

1) `start_dr_arctime`: t_r (in seconds) of the first data point.

2) `end_dr_arctime`: t_r (in seconds) of the final data point.

3) `n_data_pts_DR`: number of data points. Must be a positive integer. If equal to 1, `end_dr_arctime` is ignored. Decrease this for fast result with fewer evaluation points.

4) `plot_drscan`: If True, the `Python` program will produce a plot of $(I_{th}$ v.s $t_r/t_b)$. The user must exit the plot to obtain the data in an output file. This must be set to False for grid job submission.

Output file: `thresh_v_trotb.txt`

3.2 THRESHOLD mode

Number of argument: 3

Command: `python $DIST_BASE_DIR/bsim/bbu/test_run.py  N  fn  output_dir`

Compute the I_{th} for `N` times with a fixed lattice, each time with a (random) set of HOM assignment. It is recommended to set `N` = 1 and use external parallel computation. `fn` is the string used to distinguish between the simulation runs, and can be set to the job number (`$JOB_ID`) during grid submission.

`Python` Parameters:

(1) `random_homs`: If True, the original HOM assignment to the lattice will be over-written by a HOM assignment file randomly chosen from "`hom_dir`". The simulation result (I_{th} found) will be generally different over multiple runs. If False, the program will be seeking a file in the working directory (the directory where the user calls the `Python` program) named "`assignHOMs.bmad`" in attempt to over-write the original assignment. If "`assignHOMs.bmad`" is not present, the program will bomb.

(2) `hom_dir`: The exact path of the HOM assignment files to be randomly assigned. Each file must be named "`cavity_I.dat`", in which `I` is a positive integer.

Output file: (1) `bbu_threshold_N_fn.dat`, which includes the I_{th}s, if found, for all N runs. If not found, the numbers are the final test currents which is stable for each of the N runs.

(2) `HOMassignment_N_fn.dat` which includes the the HOM assignment scheme for all N runs.

3.3 PHASE_SCAN mode

Number of argument: 1

Command: `python $DIST_BASE_DIR/bsim/bbu/test_run.py PHASE`

Obtain (I_{th} v.s. ϕ) for a lattice with a varying horizontal phase-advance ϕ. The phase advance is changed via a zero-length first-order `Taylor` element (See Bmad manual for detail.) named "taylorW", which is equivalent to a 2x2 transfer matrix $M(\phi)$ in the horizontal phase space:

$$M(\phi) = \begin{pmatrix} (\cos\phi + \alpha_x \sin\phi) & \beta_x \sin\phi \\ \gamma_x \sin\phi & (\cos\phi - \alpha_x \sin\phi) \end{pmatrix}$$

The user must include "taylorW" in the lattice in order to run the phase_scan mode. To preserve the beam optics of the original lattice, the Twiss parameters (β_x, α_x) at where taylorW locates must be extracted from the lattice, and set in `.../bsim/bbu/python/bbu/phase_scan.py`. Due to periodicity, ϕ is only needs to be scanned from 0 to 2π.

`Python` Parameters:

1) `start_phase`: ϕ of the first data point for `n_data_pts_PHASE` $>= 2$. Default is zero.

2) `end_phase`: ϕ of the final data point for `n_data_pts_PHASE` $>= 2$. Default is 2π.

3) `n_data_pts_PHASE`: number of data points. Must be a positive integer. If equal to 1, only the input `PHASE` is scanned. If $>= 2$, the input `PHASE` is ignored.

4) `ONE_phase`: will store the input value of `PHASE`. No need to modify.

5) `plot_phase_scan`: If True, the `Python` program will produce a plot of (I_{th} v.s ϕ). The user must exit the plot to obtain the data. This must be set to False for grid jobs.

Output file: `thresh_v_phase_PHASE.txt`

3.4 PHASE_SCAN_XY mode

Number of argument: 2

Command: `python3 .../bsim/bbu/test_run.py PHASE_X PHASE_Y`

(1) Decoupled case: Obtain (I_{th} v.s. (ϕ_x, ϕ_y)) for a lattice with varying phase-advances in both X and Y (optics decoupled).

(2) Coupled case: Obtain (I_{th} v.s. (ϕ_1, ϕ_2)) for a lattice with two varying phases with X-Y coupling.

Similar to the PHASE_SCAN mode, talylorW is introduced, but this time as a 4x4 matrix:

$$T_{decoupled}(\phi_x, \phi_y) = \begin{pmatrix} M_{x\leftarrow x}(\phi_x) & \mathbf{0} \\ \mathbf{0} & M_{y\leftarrow y}(\phi_y) \end{pmatrix}$$

$$T_{coupled}(\phi_1,\phi_2) = \begin{pmatrix} \mathbf{0} & M_{x\leftarrow y}(\phi_1) \\ M_{y\leftarrow x}(\phi_2) & \mathbf{0} \end{pmatrix}$$

, in which $M(\phi)$ is a the 2x2 transfer matrix in terms of the Twiss parameters:

$$M_{1\leftarrow 0}(\phi) = \begin{pmatrix} \sqrt{\frac{\beta_1}{\beta_0}}(\cos\phi + \alpha_0 \sin\phi) & \sqrt{\beta_1\beta_0}\sin\phi \\ \frac{1}{\sqrt{\beta_1\beta_0}}[(\alpha_0-\alpha_1)\cos\phi - (1+\alpha_0\alpha_1)\sin\phi] & \sqrt{\frac{\beta_1}{\beta_0}}(\cos\phi - \alpha_1\sin\phi) \end{pmatrix}$$

For the decoupled case, the two phases are the conventional transverse phase advances. If `PHASE_Y = 0`, this is equivalent to the phase scan mode. For the coupled case, the input argument `PHASE_X` is used as ϕ_1, and `PHASE_Y` is used as ϕ_2. To use either the decoupled or coupled case, the user must extract the Twiss parameters (β_x, α_x, β_y, α_y) at where taylorW locates, and set them in `.../bsim/bbu/python/bbu/phase_scan.py`.

For a complete scan of over the two phases (both from 0 to 2π), parallel computation is recommended.

`Python` Parameters:

1) `phase_x`: will store the input value of `PHASE_X`. No need to modify.

2) `phase_y`: will store the input value of `PHASE_Y`. No need to modify.

3) `xy_coupled`: (important) 0 for the decoupled case; 1 for the coupled case.

Output file: `thresh_v_phase_PHASE_X_PHASE_Y.txt`

4 Important files

4.1 BBU Fortran codes

The files with core computation.

1)`$DIST_BASE_DIR/bsim/bbu/bbu_program.f90`: Main BBU program.

2)`$DIST_BASE_DIR/bsim/code/bbu_track_mod.f90`: BBU tracking and computation modules.

4.2 Python main user interface

The Python wrapper which interacts with the Fortran core.

1) `$DIST_BASE_DIR/bsim/bbu/test_run.py`: The TOP interface file which defines all BBU `Fortran` and `Python` parameters (to be modified by the user). When run, a temporary directory is created, which contains `bbu.init` and associated filed to run the `Fortran` core for multiple times. The directory is cleaned up at the end of the Python program. The user can deliberately terminate the `Python` program to investigate the temporary files (for debugging).

4.3 Python package

These files are for intermediate organization and communication between `test_run.py` and the `Fortran` core.

(Location: `.../bsim/bbu/python/bbu/`)(Make sure `.../bsim/bbu/pyhton/` is included in the environment variable `$PYTHONPATH`.)

1) `bbu_main.py`: Parse results from the `Fortran` code. Calls other `Python` codes.

2) `find_threshold.py`: Calculates new test current (or reference charge). Prepare temporary files to run the `Fortran` core.

3) `drscan.py`: Prepares drscan files (`lat2.lat`) and plot.

4) `phase_scan.py`: Prepares phase_scan (or phase_xy_scan) files (`lat2.lat`) and plot.

4.4 Others

Potentially useful files.

1) `collect_thresholds.py`: this `Python` program summarizes the calculated threshold currents stored in the local "bbu_thresholds_*" output files. The output file is bbu_combined_thresholds.txt.

2) `assignHOMs.bmad`: If this file exists in the directory which the user calls `test_run.py` (not necessarily the directory where `test_run.py` locates), the HOM assignment from the original lattice will be over-written, unless the user has specified `py_par['random_homs']` to be True.

References

[1] David Sagan. *The Bmad Reference Manual*, *https://www.classe.cornell.edu/bmad/manual.html*

[2] G.H. Hoffstaetter, I.V. Bazarov, *Beam-Breakup Instability Theory for Energy Recovery Linacs*, Phys. Rev. ST-AB **7**, 054401 (2004).

[3] W. Lou, G.H. Hoffstaetter, *BBeam breakup current limit in multiturn energy recovery linear accelerators*, Phys. Rev. ST-AB **22**, 112801 (2019).