Basic longitudinal structure of PyHEADTAIL

Helga Timko,
Alexandre Lasheen, Danilo Quartullo

PYlongitudinal v 1.1
Basic Functionalities

Longitudinal tracker

E.O.M. as presented previously

Acceleration, multiple RF stations, low-beta solution

Impedance (see Alex' talk)

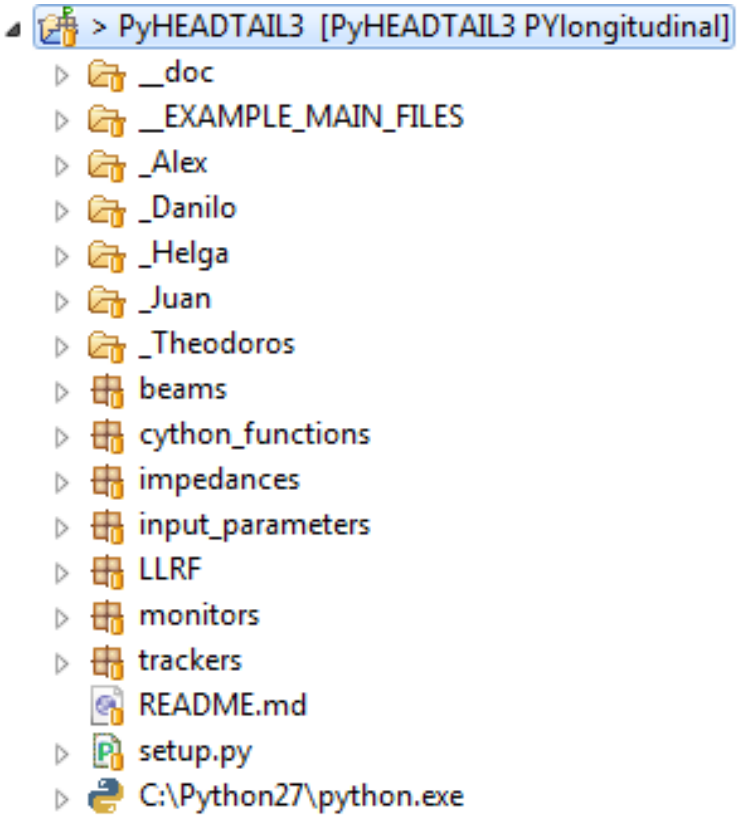
Input-output in 3 different units

Longitudinal statistics and plots

LLRF RF phase noise

To be extended with feedbacks

Documentation ('gh-pages' branch)



Versions and naming convention

PYlongitudinal Longitudinal "master" branch

Commits Have to be tested and transparent

Are given version numbers, v1.1.1

First digit	Major change (e.g. single to multi-bunch)
Second digit	Minor improvement; recommended to be downloaded
Third digit	Smaller bug fixes; you can continue with your previous version if it doesn't affect your work

Longitudinal tracker

longitudinal_tracker

```
'''
**Module containing all the elements to track the beam in the longitudinal plane.**

:Authors: **Danilo Quartullo**, **Helga Timko**, **Adrian Oeftiger**, **Alexandre Lasheen**
'''

from __future__ import division
import numpy as np
from scipy.constants import c


class RingAndRFSection(object):
    '''...'''

    def __init__(self, rf_params, solver='full'):...


    def kick(self, beam):...


    def kick_acceleration(self, beam):...


    def drift(self, beam):...


    def track(self, beam):...


class LinearMap(object):

    '''...'''

    def __init__(self, GeneralParameters, Qs):...


    def track(self, beam):...
```

We ended up with the kicks and drift being part of the RingAndRFSection, as the class is already defined in a way that gives the most general combination of the kicks and the drift that is implemented

Longitudinal tracker equations

According to the E.O.M. presented in the previous meeting...

```
    def kick(self, beam):
        '''..

        for i in range(self.n_rf):
            beam.dE += self.voltage[i,self.counter[0]] * \
                       np.sin(self.harmonic[i,self.counter[0]] *
                              beam.theta + self.phi_offset[i,self.counter[0]])


    def kick_acceleration(self, beam):
        '''..

        beam.dE += self.acceleration_kick[self.counter[0]]


    def drift(self, beam):
        '''..

        if self.solver == 'full':
            beam.theta = self.beta_ratio[self.counter[0]] * beam.theta \
                         + 2 * np.pi * (1 / (1 - self.rf_params.eta_tracking(beam.delta) *
                                             beam.delta) - 1) * self.length_ratio
        elif self.solver == 'simple':
            beam.theta = self.beta_ratio[self.counter[0]] *beam.theta \
                         + 2 * np.pi * self.eta_0[self.counter[0]] \
                         * beam.delta * self.length_ratio
        else:
            raise RuntimeError("ERROR: Choice of longitudinal solver not \
                               recognized! Aborting...")
```

Longitudinal tracker equations

According to the E.O.M. presented in the previous meeting...

```
    def kick(self, beam):
        '''..

        for i in range(self.n_rf):
            beam.dE += self.voltage[i,self.counter[0]] * \
                       np.sin(self.harmonic[i,self.counter[0]] *
                              beam.theta + self.phi_offset[i,self.counter[0]])


    def kick_acceleration(self, beam):
        '''..

        beam.dE += self.acceleration_kick[self.counter[0]]


    def drift(self, beam):
        '''..

        if self.solver == 'full':
            beam.theta = self.beta_ratio[self.counter[0]] * beam.theta \
                         + 2 * np.pi * (1 / (1 - self.rf_params.eta_tracking(beam.delta) *
                                             beam.delta) - 1) * self.length_ratio
        elif self.solver == 'simple':
            beam.theta = self.beta_ratio[self.counter[0]] *beam.theta \
                         + 2 * np.pi * self.eta_0[self.counter[0]] \
                         * beam.delta * self.length_ratio
        else:
            raise RuntimeError("ERROR: Choice of longitudinal solver not \
                               recognized! Aborting...")
```

Full eta calculation

Longitudinal tracker equations

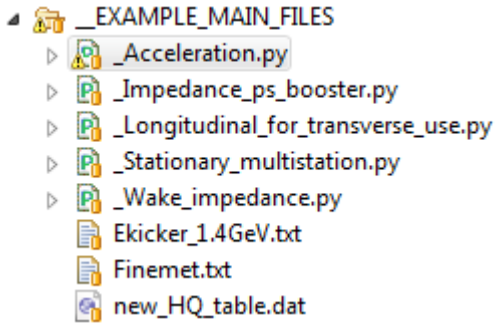
According to the E.O.M. presented in the previous meeting...

```
    def kick(self, beam):
        '''..

        for i in range(self.n_rf):
            beam.dE += self.voltage[i,self.counter[0]] * \
                       np.sin(self.harmonic[i,self.counter[0]] *
                              beam.theta + self.phi_offset[i,self.counter[0]])


    def kick_acceleration(self, beam):
        '''..

        beam.dE += self.acceleration_kick[self.counter[0]]


    def drift(self, beam):
        '''..

        if self.solver == 'full':
            beam.theta = self.beta_ratio[self.counter[0]] * beam.theta \
                         + 2 * np.pi * (1 / (1 - self.rf_params.eta_tracking(beam.delta) *
                                             beam.delta) - 1) * self.length_ratio
        elif self.solver == 'simple':
            beam.theta = self.beta_ratio[self.counter[0]] *beam.theta \
                         + 2 * np.pi * self.eta_0[self.counter[0]]
                         * beam.delta * self.length_ratio
        else:
            raise RuntimeError("ERROR: Choice of longitudinal solver not \
                                recognized! Aborting...")
```

Full eta calculation

Zeroth order eta

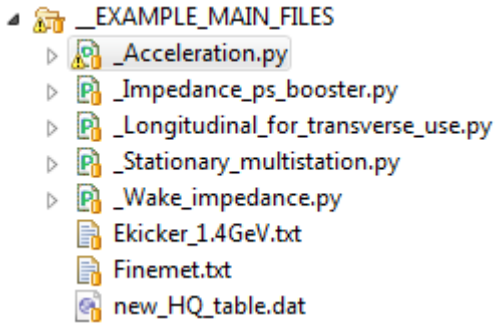
__EXAMPLE_MAIN_FILES
Minimal longitudinal part



_Longitudinal_for_transverse_use

```
# Simulation setup -----------------------------------------------------------
print "Setting up the simulation..."
print ""


# Define general parameters
general_params = GeneralParameters(N_t, C, alpha, p_s, 'proton')

# Define longitudinal tracker for transverse use
long_tracker = LinearMap(general_params, Qs)

print "General and RF parameters set..."


# Define beam and distribution
beam = Beam(general_params, N_p, N_b)
beam.theta = s_theta*np.random.randn(N_p)
beam.dE = s_dE*np.random.randn(N_p)

print "Beam set and distribution generated..."


# Need slices for the Gaussian fit; slice for the first plot
slice_beam = Slices(100)
slice_beam.track(beam)

# Define what to save in file
bunchmonitor = BunchMonitor('output_data', N_t+1, statistics = "Longitudinal", long_gaussian_fit = "On")

print "Statistics set..."


# Accelerator map
map_ = [long_tracker] + [slice_beam] # No intensity effects, no aperture limitations
print "Map set"
print ""
```

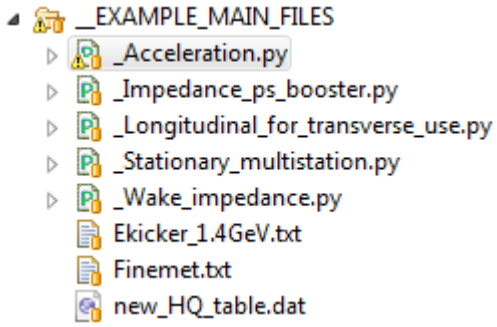
__EXAMPLE_MAIN_FILES
Acceleration



_Acceleration

```
# Simulation setup ------------------------------------------------------------
print "Setting up the simulation..."
print ""


# Define general parameters
general_params = GeneralParameters(N_t, C, alpha, np.linspace(p_i, p_f, 2002),
                                   'proton')

# Define RF station parameters and corresponding tracker
rf_params = RFSectionParameters(general_params, 1, 1, h, V, dphi)
long_tracker = RingAndRFSection(rf_params)

print "General and RF parameters set..."


# Define beam and distribution
beam = Beam(general_params, N_p, N_b)
longitudinal_gaussian_matched(general_params, rf_params, beam, tau_0,
                              unit='ns', reinsertion = 'on')

print "Beam set and distribution generated..."


# Need slices for the Gaussian fit; slice for the first plot
slice_beam = Slices(100)
slice_beam.track(beam)

# Define what to save in file
bunchmonitor = BunchMonitor('output_data', N_t+1, statistics = "Longitudinal", long_gaussian_fit = "On")

print "Statistics set..."


# Accelerator map
map_ = [long_tracker] + [slice_beam] # No intensity effects, no aperture limitations
print "Map set"
print ""
```

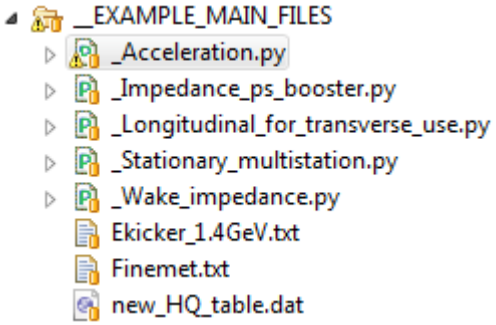
__EXAMPLE_MAIN_FILES
Multiple RF stations



_Stationary_multistation

```
# Simulation setup ------------------------------------------------------------
print "Setting up the simulation..."
print ""


# Define general parameters containing data for both RF stations
general_params = GeneralParameters(N_t, [0.3*C, 0.7*C], [[alpha], [alpha]],
                                   [p_s*np.ones(N_t+1), p_s*np.ones(N_t+1)],
                                   'proton', number_of_sections = 2)

# Define RF station parameters and corresponding tracker
rf_params_1 = RFSectionParameters(general_params, 1, 1, h, V1, dphi)
long_tracker_1 = RingAndRFSection(rf_params_1)

rf_params_2 = RFSectionParameters(general_params, 2, 1, h, V2, dphi)
long_tracker_2 = RingAndRFSection(rf_params_2)

# Define full voltage over one turn and a corresponding "overall" set of
#parameters, which is used for the separatrix (in plotting and losses)
Vtot = total_voltage([rf_params_1, rf_params_2])
rf_params_tot = RFSectionParameters(general_params, 1, 1, h, Vtot, dphi)
long_tracker_tot = RingAndRFSection(rf_params_tot)
beam_dummy = Beam(general_params, 0, N_b)
print "General and RF parameters set..."
print Vtot

# Define beam and distribution
beam = Beam(general_params, N_p, N_b)
longitudinal_gaussian_matched(general_params, rf_params_tot, beam, tau_0,
                              unit='ns', reinsertion = 'on')

print "Beam set and distribution generated..."
```

__EXAMPLE_MAIN_FILES
Multiple RF stations



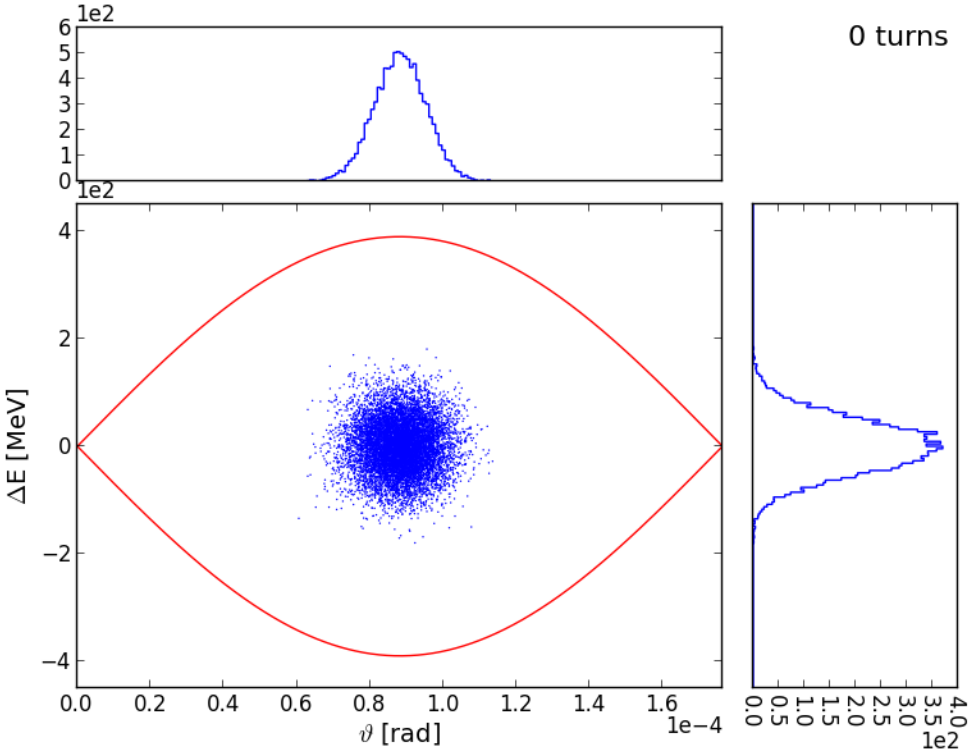
_Stationary_multistation

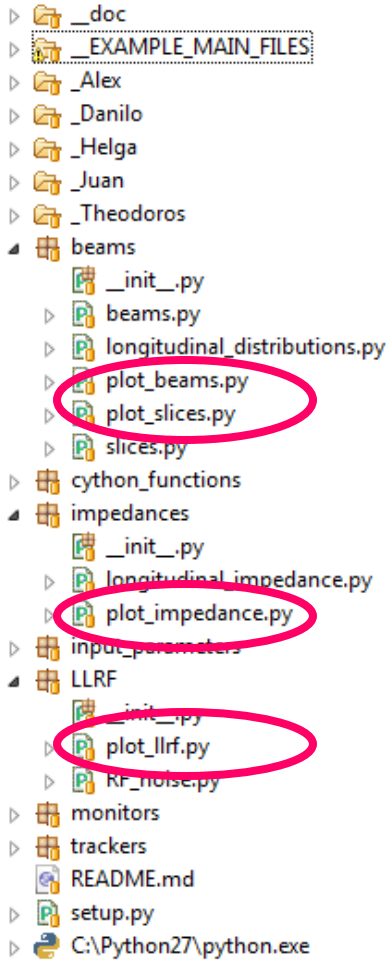
```
# Simulation setup ------------------------------------------------------------
print "Setting up the simulation..."
print ""


# Define general parameters containing data for both RF stations
general_params = GeneralParameters(N_t, [0.3*C, 0.7*C], [[alpha], [alpha]],
                                   [p_s*np.ones(N_t+1), p_s*np.ones(N_t+1)],
                                   'proton', number_of_sections = 2)

# Define RF station parameters and corresponding tracker
rf_params_1 = RFSectionParameters(general_params, 1, 1, h, V1, dphi)
long_tracker_1 = RingAndRFSection(rf_params_1)

rf_params_2 = RFSectionParameters(general_params, 2, 1, h, V2, dphi)
long_tracker_2 = RingAndRFSection(rf_params_2)

# Define full voltage over one turn and a corresponding "overall" set of
#parameters which is used for the separatrix (in plotting and losses)
Vtot = total_voltage([rf_params_1, rf_params_2])
rf_params_tot = RFSectionParameters(general_params, 1, 1, h, Vtot, dphi)
long_tracker_tot = RingAndRFSection(rf_params_tot)
beam_dummy = Beam(general_params, 0, N_b)
print "General and RF parameters set..."
print Vtot

# Define beam and distribution
beam = Beam(general_params, N_p, N_b)
longitudinal_gaussian_matched(general_params, rf_params_tot, beam, tau_0,
                              unit='ns', reinsertion = 'on')

print "Beam set and distribution generated..."
```

Temporary solution

Some basic output

Longitudinal phase space, bunch length… etc.





Sphinx documentation

Edit .rst files

Modules to be documented

Comment python files

See next slide

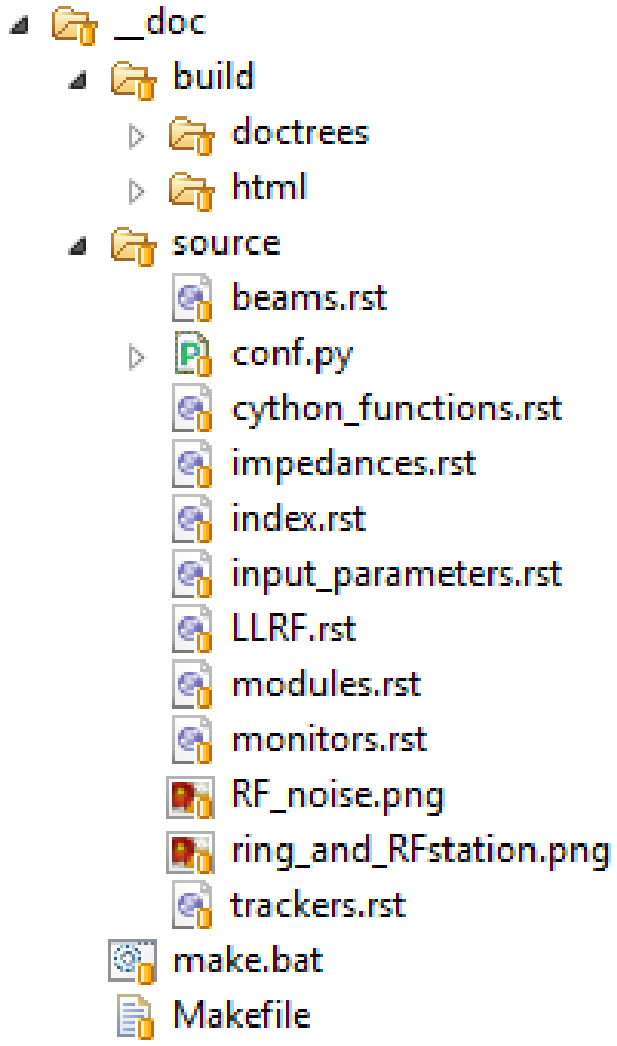
Compile and create '**build**' folder

Run 'make html' in '__doc' folder

Update the gh-pages branch

Copy the content of 'html' folder

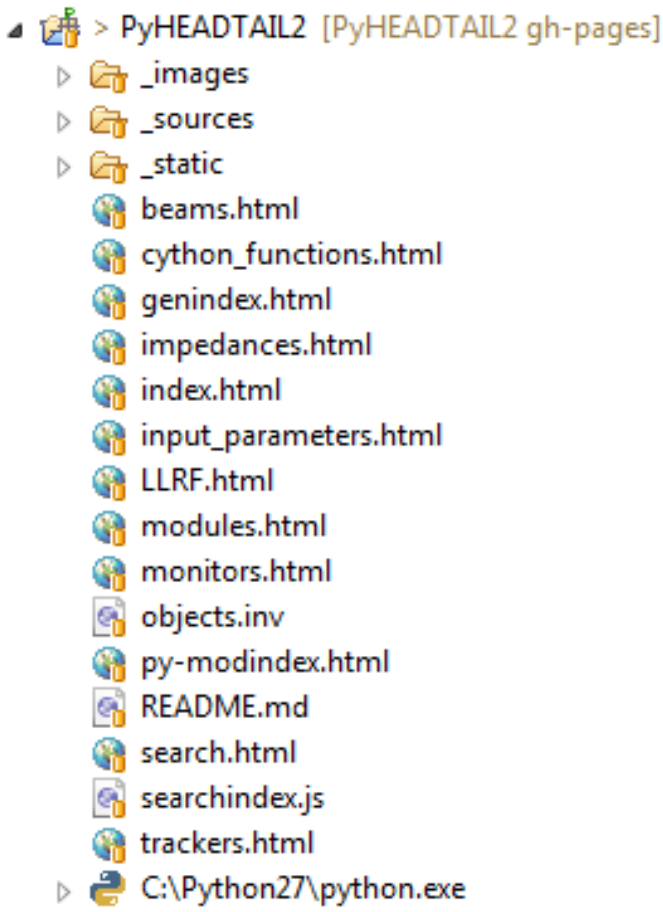
For more info, ask Alex
Check out the cheat sheet



Sphinx documentation – Commenting python files

.gitignore | Welcome to PyHEADTAIL's documentation! — PyHEADTAIL embryo documentation | longitudinal_tracker

```
'''
**Module containing all the elements to track the beam in the longitudinal plane.**

:Authors: **Danilo Quartullo**, **Helga Timko**, **Adrian Oeftiger**, **Alexandre Lasheen**
'''

from __future__ import division
import numpy as np
from scipy.constants import c


class RingAndRFSection(object):
    '''
    *Definition of an RF station and part of the ring until the next station,
    see figure.*

    .. image:: ring_and_RFstation.png
        :align: center
        :width: 600
        :height: 600

    *The time step is fixed to be one turn, but the tracking can consist of
    multiple RingAndRFSection objects. In this case, the user should make sure
    that the lengths of the stations sum up exactly to the circumference or use
    the FullRingAndRF object in order to let the code pre-process the parameters.
    Each RF station may contain several RF harmonic systems which are considered
    to be in the same location. First, a kick from the cavity voltage(s) is applied,
    then an accelerating kick in case the momentum program presents variations,
    and finally a drift kick between stations.*
    '''
```



Sphinx documentation – Commenting python files

.gitignore | Welcome to PyHEADTAIL's documentation! — PyHEADTAIL embryo documentation | longitudinal_tracker

```
'''
**Module containing all the elements to track the beam in the longitudinal plane.**

:Authors: **Danilo Quartullo**, **Helga Timko**, **Adrian Oeftiger**, **Alexandre Lasheen**
'''

from __future__ import division
import numpy as np
from scipy.constants import c


class RingAndRFSection(object):
    '''
    *Definition of an RF station and part of the ring until the next station,
    see figure.*

    .. image:: ring_and_RFstation.png
        :align: center
        :width: 600
        :height: 600

    *The time step is fixed to be one turn, but the tracking can consist of
    multiple RingAndRFSection objects. In this case, the user should make sure
    that the lengths of the stations sum up exactly to the circumference or use
    the FullRingAndRF object in order to let the code pre-process the parameters.
    Each RF station may contain several RF harmonic systems which are considered
    to be in the same location. First, a kick from the cavity voltage(s) is applied,
    then an accelerating kick in case the momentum program presents variations,
    and finally a drift kick between stations.*
    '''
```

HEADER, AUTHORS

Sphinx documentation – Commenting python files

.gitignore | Welcome to PyHEADTAIL's documentation! — PyHEADTAIL embryo documentation | longitudinal_tracker

```
'''
**Module containing all the elements to track the beam in the longitudinal plane.**

:Authors: **Danilo Quartullo**, **Helga Timko**, **Adrian Oeftiger**, **Alexandre Lasheen**
'''

from __future__ import division
import numpy as np
from scipy.constants import c


class RingAndRFSection(object):
    '''
    *Definition of an RF station and part of the ring until the next station,
    see figure.*

    .. image:: ring_and_RFstation.png
        :align: center
        :width: 600
        :height: 600

    *The time step is fixed to be one turn, but the tracking can consist of
    multiple RingAndRFSection objects. In this case, the user should make sure
    that the lengths of the stations sum up exactly to the circumference or use
    the FullRingAndRF object in order to let the code pre-process the parameters.
    Each RF station may contain several RF harmonic systems which are considered
    to be in the same location. First, a kick from the cavity voltage(s) is applied,
    then an accelerating kick in case the momentum program presents variations,
    and finally a drift kick between stations.*
    '''
```

HEADER, AUTHORS

CLASSES and FUNCTIONS

Sphinx documentation – Commenting python files

.gitignore | Welcome to PyHEADTAIL's documentation! — PyHEADTAIL embryo documentation | longitudinal_tracker

```
'''
**Module containing all the elements to track the beam in the longitudinal plane.**

:Authors: **Danilo Quartullo**, **Helga Timko**, **Adrian Oeftiger**, **Alexandre Lasheen**
'''

from __future__ import division
import numpy as np
from scipy.constants import c


class RingAndRFSection(object):
    '''
    *Definition of an RF station and part of the ring until the next station,
    see figure.*

    .. image:: ring_and_RFstation.png
        :align: center
        :width: 600
        :height: 600

    *The time step is fixed to be one turn, but the tracking can consist of
    multiple RingAndRFSection objects. In this case, the user should make sure
    that the lengths of the stations sum up exactly to the circumference or use
    the FullRingAndRF object in order to let the code pre-process the parameters.
    Each RF station may contain several RF harmonic systems which are considered
    to be in the same location. First, a kick from the cavity voltage(s) is applied,
    then an accelerating kick in case the momentum program presents variations,
    and finally a drift kick between stations.*
    '''
```

HEADER, AUTHORS

CLASSES and FUNCTIONS

INSERT IMAGE

Sphinx documentation – Commenting python files

```
def kick(self, beam):
    '''
    *The Kick represents the kick(s) by an RF station at a certain position
    of the ring. The kicks are summed over the different harmonic RF systems
    in the station. The cavity phase can be shifted by the user via phi_offset.
    The increment in energy is given by the discrete equation of motion:*

    .. math::
        \Delta E_{n+1} = \Delta E_n + \sum_{j=0}^{n_{RF}}{V_{j,n}\,\sin{\\left(h_{j,n}\,\\theta + \phi_{j,n}\\right)}}

    '''

    for i in range(self.n_rf):
        beam.dE += self.voltage[i,self.counter[0]] * \
                   np.sin(self.harmonic[i,self.counter[0]] *
                          beam.theta + self.phi_offset[i,self.counter[0]])
```

```
def __init__(self, rf_params, solver='full'):

    #: | *Choice of solver for the drift*
    #: | *Use 'full' for full eta solver*
    #: | *Use 'simple' for 0th order eta solver*
    self.solver = solver

    #: | *Counter to keep track of time step (used in momentum and voltage)*
    self.counter = rf_params.counter

    #: | *Import RF section parameters for RF kick*
    #: | *Length ratio between drift and ring circumference*
    #: | :math:`: \quad \frac{L}{C}`
    self.length_ratio = rf_params.length_ratio
    #: | *Harmonic number list* :math:`: \quad h_{j,n}`
    self.harmonic = rf_params.harmonic
    #: | *Voltage program list in [V]* :math:`: \quad V_{j,n}`
    self.voltage = rf_params.voltage
    #: | *Phase offset list in [rad]* :math:`: \quad \phi_{j,n}`
    self.phi_offset = rf_params.phi_offset
    #: | *Number of RF systems in the RF station* :math:`: \quad n_{RF}`
    self.n_rf = rf_params.n_rf
```

Sphinx documentation – Commenting python files

```
def kick(self, beam):
    '''
    *The Kick represents the kick(s) by an RF station at a certain position
    of the ring. The kicks are summed over the different harmonic RF systems
    in the station. The cavity phase can be shifted by the user via phi_offset.
    The increment in energy is given by the discrete equation of motion:*

    .. math::
        \Delta E_{n+1} = \Delta E_n + \sum_{j=0}^{n_{RF}}{V_{j,n}\,\sin{\\left(h_{j,n}\,\\theta + \phi_{j,n}\\right)}}

    '''

    for i in range(self.n_rf):
        beam.dE += self.voltage[i,self.counter[0]] * \
                   np.sin(self.harmonic[i,self.counter[0]] *
                          beam.theta + self.phi_offset[i,self.counter[0]])
```

MATH in function or class

```
def __init__(self, rf_params, solver='full'):

    #: | *Choice of solver for the drift*
    #: | *Use 'full' for full eta solver*
    #: | *Use 'simple' for 0th order eta solver*
    self.solver = solver

    #: | *Counter to keep track of time step (used in momentum and voltage)*
    self.counter = rf_params.counter

    #: | *Import RF section parameters for RF kick*
    #: | *Length ratio between drift and ring circumference*
    #: | :math:`: \quad \frac{L}{C}`
    self.length_ratio = rf_params.length_ratio
    #: | *Harmonic number list* :math:`: \quad h_{j,n}`
    self.harmonic = rf_params.harmonic
    #: | *Voltage program list in [V]* :math:`: \quad V_{j,n}`
    self.voltage = rf_params.voltage
    #: | *Phase offset list in [rad]* :math:`: \quad \phi_{j,n}`
    self.phi_offset = rf_params.phi_offset
    #: | *Number of RF systems in the RF station* :math:`: \quad n_{RF}`
    self.n_rf = rf_params.n_rf
```

Sphinx documentation – Commenting python files

```python
def kick(self, beam):
    '''
    *The Kick represents the kick(s) by an RF station at a certain position
    of the ring. The kicks are summed over the different harmonic RF systems
    in the station. The cavity phase can be shifted by the user via phi_offset.
    The increment in energy is given by the discrete equation of motion:*

    .. math::
        \Delta E_{n+1} = \Delta E_n + \sum_{j=0}^{n_{RF}}{V_{j,n}\,\sin{\\left(h_{j,n}\,\\theta + \phi_{j,n}\\right)}}

    '''

    for i in range(self.n_rf):
        beam.dE += self.voltage[i,self.counter[0]] * \
                   np.sin(self.harmonic[i,self.counter[0]] *
                          beam.theta + self.phi_offset[i,self.counter[0]])
```

MATH in function or class

```python
def __init__(self, rf_params, solver='full'):

    #: | *Choice of solver for the drift*
    #: | *Use 'full' for full eta solver*
    #: | *Use 'simple' for 0th order eta solver*
    self.solver = solver

    #: | *Counter to keep track of time step (used in momentum and voltage)*
    self.counter = rf_params.counter

    #: | *Import RF section parameters for RF kick*
    #: | *Length ratio between drift and ring circumference*
    #: | :math:`: \quad \frac{L}{C}`
    self.length_ratio = rf_params.length_ratio
    #: | *Harmonic number list* :math:`: \quad h_{j,n}`
    self.harmonic = rf_params.harmonic
    #: | *Voltage program list in [V]* :math:`: \quad V_{j,n}`
    self.voltage = rf_params.voltage
    #: | *Phase offset list in [rad]* :math:`: \quad \phi_{j,n}`
    self.phi_offset = rf_params.phi_offset
    #: | *Number of RF systems in the RF station* :math:`: \quad n_{RF}`
    self.n_rf = rf_params.n_rf
```

PROPERTIES

Sphinx documentation – Commenting python files

```python
def kick(self, beam):
    '''
    *The Kick represents the kick(s) by an RF station at a certain position
    of the ring. The kicks are summed over the different harmonic RF systems
    in the station. The cavity phase can be shifted by the user via phi_offset.
    The increment in energy is given by the discrete equation of motion:*

    .. math::
        \Delta E_{n+1} = \Delta E_n + \sum_{j=0}^{n_{RF}}{V_{j,n}\,\sin{\\left(h_{j,n}\,\\theta + \phi_{j,n}\\right)}}

    '''

    for i in range(self.n_rf):
        beam.dE += self.voltage[i,self.counter[0]] * \
                   np.sin(self.harmonic[i,self.counter[0]] *
                          beam.theta + self.phi_offset[i,self.counter[0]])
```

MATH in function or class

```python
def __init__(self, rf_params, solver='full'):

    #: | *Choice of solver for the drift*
    #: | *Use 'full' for full eta solver*
    #: | *Use 'simple' for 0th order eta solver*
    self.solver = solver

    #: | *Counter to keep track of time step (used in momentum and voltage)*
    self.counter = rf_params.counter

    #: | *Import RF section parameters for RF kick*
    #: | *Length ratio between drift and ring circumference*
    #: | :math:`: \quad \frac{L}{C}`
    self.length_ratio = rf_params.length_ratio
    #: | *Harmonic number list* :math:`: \quad h_{j,n}`
    self.harmonic = rf_params.harmonic
    #: | *Voltage program list in [V]* :math:`: \quad V_{j,n}`
    self.voltage = rf_params.voltage
    #: | *Phase offset list in [rad]* :math:`: \quad \phi_{j,n}`
    self.phi_offset = rf_params.phi_offset
    #: | *Number of RF systems in the RF station* :math:`: \quad n_{RF}`
    self.n_rf = rf_params.n_rf
```

PROPERTIES

USE | for new line
USE * for italics

Sphinx documentation – Commenting python files

```python
def kick(self, beam):
    '''
    *The Kick represents the kick(s) by an RF station at a certain position
    of the ring. The kicks are summed over the different harmonic RF systems
    in the station. The cavity phase can be shifted by the user via phi_offset.
    The increment in energy is given by the discrete equation of motion:*

    .. math::
        \Delta E_{n+1} = \Delta E_n + \sum_{j=0}^{n_{RF}}{V_{j,n}\,\sin{\\left(h_{j,n}\,\\theta + \phi_{j,n}\\right)}}

    '''

    for i in range(self.n_rf):
        beam.dE += self.voltage[i,self.counter[0]] * \
                   np.sin(self.harmonic[i,self.counter[0]] *
                          beam.theta + self.phi_offset[i,self.counter[0]])
```

MATH in function or class

```python
def __init__(self, rf_params, solver='full'):

    #: | *Choice of solver for the drift*
    #: | *Use 'full' for full eta solver*
    #: | *Use 'simple' for 0th order eta solver*
    self.solver = solver

    #: | *Counter to keep track of time step (used in momentum and voltage)*
    self.counter = rf_params.counter

    #: | *Import RF section parameters for RF kick*
    #: | *Length ratio between drift and ring circumference*
    #: | :math:`: \quad \frac{L}{C}`
    self.length_ratio = rf_params.length_ratio
    #: | *Harmonic number list* :math:`: \quad h_{j,n}`
    self.harmonic = rf_params.harmonic
    #: | *Voltage program list in [V]* :math:`: \quad V_{j,n}`
    self.voltage = rf_params.voltage
    #: | *Phase offset list in [rad]* :math:`: \quad \phi_{j,n}`
    self.phi_offset = rf_params.phi_offset
    #: | *Number of RF systems in the RF station* :math:`: \quad n_{RF}`
    self.n_rf = rf_params.n_rf
```

PROPERTIES

USE | for new line
USE * for italics

MATH in property

Conclusions

PYlongitudinal is ready to use

The tracker	Has been benchmarked
Version numbers	Only tested features can be added to the code to make all modifications as transparent as possible to the user

Contact Danilo for any modifications

What's left to do

Longitudinal_utilities.py

- Methods for separatrix, hamiltonian, total voltage, etc. are presently defined for single RF only
  - Needs to be extended for multi-RF, multi-station cases

Generating matched bunch distributions

- Only single RF, Gaussian distribution, w/o intensity effects
  - Matching w/ adiabatic increase of impedance already possible
  - Matching w/ Hamiltonian needs to be implemented (Theodoros)
  - Implementing other distributions (parabolic, etc.)

LLRF Planned extension w/ feedbacks

Slices Need to agree on this together (see Alex' talk)

Outlook

Multi-bunch case Danilo, from September (?)

Merge transverse and longitudinal codes

Outlook

Multi-bunch case Danilo, from September (?)

Merge transverse and longitudinal codes

Agree on the common transverse version

Outlook

Multi-bunch case Danilo, from September (?)

Merge transverse and longitudinal codes

- **Agree on the common transverse version**
- **Meanwhile develop transverse and longitudinal parts such that they stay compatible**

Outlook

Multi-bunch case Danilo, from September (?)

Merge transverse and longitudinal codes

- **Agree on the common transverse version**
- **Meanwhile develop transverse and longitudinal parts such that they stay compatible**
- **Not wait too long to update the master branch with a merge of the two**

Outlook

Multi-bunch case Danilo, from September (?)

Merge transverse and longitudinal codes

- **Agree on the common transverse version**
- **Meanwhile develop transverse and longitudinal parts such that they stay compatible**
- **Not wait too long to update the master branch with a merge of the two**
- **⇒ We'll have to sit down and do the final merge together!**