**Benchmarking space charge in the linear IOTA 8.2 lattice**

This notebook performs simple simulations of the linear IOTA 8.2 lattice with a fixed bunch current, and evaluates the bunch behavior in terms of envelope and emittance evolution as well as tune depression. The goal of the notebook is to verify that the proper space charge tune depression is achieved in the IOTA lattice for matching bunches with specially tailored lattice variants. This notebook, the associated scripts, and a host of other resources can be found at RadiaSoft's [rssynergia](https://github.com/radiasoft/rssynergia) github repository: <https://github.com/radiasoft/rssynergia>.

*NOTE: Lattice files may be found in the corresponding* [*ioptics*](https://github.com/radiasoft/ioptics) *repository.*

**Basic Simulation Parameters**

* IOTA v8.2 lattice with 1 integrable optics drift section (no nonlinear magnet in place)
* Sextupole strengths set to zero
* Full CHEF propagation - symplectic single particle tracking including nonlinear effects
* Each element in the lattice is sliced into 4 segments for computing dynamics.

**Basic Beam Parameters**

* 2.5 MeV proton beam
* KV distribution with total geometric emittance of 20 mm-mrad
* 4.25 mA of current, corresponding to 4.854214e+10 total protons or 1.214518e+07 p+/cm around the ring.

**Specific setup notes**

1. Be sure to have the rssynergia repo in your path
2. Adjust the dir\_66 and dir\_82 values in code block **6** to match the location of your lattice files.
3. If you do not have the watermark package, then its import may be commented out as its non-essential to running the notebook.

**Imports**

In [1]:

%matplotlib inline

%load\_ext autoreload

%autoreload 2

In [2]:

import sys, os

import numpy as np

import matplotlib.pyplot as plt

import matplotlib as mpl

import scipy

from scipy import constants

try:

import tables

except ImportError:

! pip install tables

from mpi4py import MPI

In [3]:

from rssynergia.base\_diagnostics import utils

from rssynergia.base\_diagnostics import read\_bunch

from rssynergia.base\_diagnostics import workflow

from rssynergia.base\_diagnostics import lfplot

from rssynergia.base\_diagnostics import latticework

from rssynergia.base\_diagnostics import basic\_calcs

from rssynergia.base\_diagnostics import pltbunch

from rssynergia.base\_diagnostics import elliptic\_sp

from rssynergia.base\_diagnostics import singleparticle

from rssynergia.base\_diagnostics import options

from rssynergia.base\_diagnostics import diagplot

from rssynergia.standard import standard\_beam6d

from rssynergia.elliptic import elliptic\_beam6d

#from rssynergia.semigaussian import semi\_gaussian6d

import synergia

import synergia\_workflow

**Load options and lattices**

In [4]:

#load options for SC\_test

from SC\_test\_options import opts

#================== Setting up logger and MPI comunicator ============================

#try:

#if True:

# this is the communicator object that will be used for MPI operations

comm = synergia.utils.Commxx()

myrank = comm.get\_rank()

mpisize = comm.get\_size()

verbose = opts.verbosity>0

logger = synergia.utils.Logger(0)

if myrank == 0:

print "my rank is 0"

else:

print "not rank 0"

my rank is 0

In [19]:

comm

Out[19]:

<synergia.utils.parallel\_utils.Commxx at 0x7f228788edb8>

In [10]:

#================== Load the lattice =======================

#Assumes radiasoft/ioptics and radiasoft/rssynergia share a common directory

lattices = {}

dir\_66 = '../../../ioptics/ioptics/lattices/Iota6-6/'

dir\_82 = '../../../ioptics/ioptics/lattices/Iota8-2/'

lattices['t1\_1IO\_66'] = dir\_66 + "lattice\_1IO\_center.madx" #centered t1 6.6 1IO lattice

lattices['t3\_1IO\_66'] = dir\_66 + "lattice\_1IO\_nll\_center.madx" #centered t3 6.6 1IO lattice

lattices['t1\_1IO\_82\_uncentered'] = dir\_82 + "lattice\_1IO.madx" #uncentered tier1 8.2 lattice

lattices['t1\_1IO\_82'] = dir\_82 + "lattice\_1IO\_center.madx" #this is the new tier1 8.2 (zero current) lattice

lattices['t1\_1IO\_82\_dQ\_1'] = dir\_82 + "lattice\_1IO\_dQ\_1.madx" #t1 8.2 lattice adjusted for dQ = -0.1

lattices['t1\_1IO\_82\_dQ'] = dir\_82 + "lattice\_1IO\_dQ\_2.madx" #t1 8.2 lattice adjusted for dQ = -0.2

lattices['t3\_1IO\_82'] = dir\_82 + "lattice\_1IO\_nll\_center.madx" #this is the centerted t3 8.2 lattice

lattices['t3\_1IO\_82\_dQ'] = dir\_82 + "lattice\_1IO\_nll\_dQ\_2.madx" #t3 8.2 lattice adjusted for dQ = -0.2

In [11]:

#================= Construct a Python dictionary of lattice stuff ==================

lattice\_dict = {}

for keys in lattices.keys():

lattice\_dict[keys] = {} #instantiate sub dictionary

lattice\_dict[keys]['name'] = keys

lattice\_dict[keys]['location'] = lattices[keys]

lattice\_dict[keys]['lattice'] = synergia.lattice.MadX\_reader().get\_lattice("iota", lattices[keys])

In [12]:

reference\_particle = lattice\_dict['t1\_1IO\_82\_dQ\_1']['lattice'].get\_reference\_particle()

energy = reference\_particle.get\_total\_energy()

opts.beta = reference\_particle.get\_beta()

opts.gamma = reference\_particle.get\_gamma()

#================== Setting up the options =======================

turns = 100

order = 1

nsteps\_per\_element = 4

n\_ppc = 100

opts.gridx = 32

opts.gridy = 32

opts.gridz = 1

n\_macro = n\_ppc\*opts.gridx\*opts.gridy

#n\_macro = 6\*60 #60 particles per core when running on 6 cores

dpop = 0.0

#emittances = [9.74e-7] #reduce emittance by a factor of 10

emit\_n = 3.0e-7 #0.3 mm-mrad emittance

emittances = [basic\_calcs.calc\_geometric\_emittance(emit\_n,opts.beta,opts.gamma)]

opts.emits = emittances

tval = 0.00000000001

cval = 0.01

outputdir = 'rssynergia-Benchmark-8pt5-LINEAR'

nsteps = len(lattice\_dict['t1\_1IO\_82\_dQ\_1']['lattice'].get\_elements())\*nsteps\_per\_element

opts.output\_dir = outputdir

opts.relpath = opts.output\_dir

opts.macro\_particles = n\_macro

opts.steps = nsteps

opts.steps\_per\_element = nsteps\_per\_element

workflow.make\_path(opts.output\_dir)

print >>logger, "output directory:", opts.output\_dir

#==================== Set up space charge solver ==========================

#turn off space charge

opts.spacecharge = True

requested\_stepper = opts.stepper

if opts.spacecharge:

solver = opts.solver

# space charge only works with the split operator stepper, or soelements

if (requested\_stepper != "splitoperator") and (requested\_stepper != "soelements"):

requested\_stepper = "soelements"

print "Requested stepper changed to soelements for space charge"

#force these

gridx = 32

gridy = 32

gridz = 1

grid = [gridx, gridy, gridz]

print >>logger, "grid: ", grid

#opts.comm\_divide = None

if opts.comm\_divide:

sc\_comm = synergia.utils.Commxx\_divider(opts.comm\_divide, False)

else:

sc\_comm = synergia.utils.Commxx(True)

#sc\_comm = synergia.utils.Commxx(True)

if solver == "2dopen-hockney":

coll\_operator = synergia.collective.Space\_charge\_2d\_open\_hockney(sc\_comm, grid)

elif solver == "3dopen-hockney":

# full signature for 3d\_open\_hockney constructor is

# comm, grid, long\_kicks, z\_periodic, period, grid\_entire\_period,

# nsigma

coll\_operator = synergia.collective.Space\_charge\_3d\_open\_hockney(sc\_comm, grid, opts.long\_kicks, False, 0.0, False, opts.nsigma)

elif solver == "2dbassetti-erskine":

coll\_operator = synergia.collective.Space\_charge\_2d\_bassetti\_erskine()

else:

raise RuntimeError, "requested space charge operator %s invalid. Must be either 2dopen-hockney or 3dopen-hockney"%opts.solver

print "Using space charge solver ", solver

print "Grid: ", gridx, " x ", gridy, " x ", gridz

else:

coll\_operator = synergia.simulation.Dummy\_collective\_operator("stub")

print "No space charge solver used"

#opts.use\_maps = 'none'

#now set element type and construct stepper

print "use maps for: {}".format(opts.use\_maps)

print "requested\_stepper: {}".format(requested\_stepper)

#for key in lattices.keys():

# current\_lattice = synergia.lattice.MadX\_reader().get\_lattice("iota", lattices[key])

# lattice\_dict[key]['lattice'] = latticework.set\_lattice\_element\_type(current\_lattice,opts)

#================== Setting up the stepper and lattice simulator =======================

for key in lattices.keys():

current\_lattice = lattice\_dict[key]['lattice']

lattice\_dict[key]['stepper'] = latticework.generate\_stepper(current\_lattice,coll\_operator, opts)

lattice\_dict[key]['lattice\_simulator'] = lattice\_dict[key]['stepper'].get\_lattice\_simulator()

Using space charge solver 2dopen-hockney

Grid: 32 x 32 x 1

use maps for: none

requested\_stepper: splitoperator

Using split-operator stepper with 1368 steps/turn

Using split-operator stepper with 1368 steps/turn

Using split-operator stepper with 1368 steps/turn

Using split-operator stepper with 1368 steps/turn

Using split-operator stepper with 1368 steps/turn

Using split-operator stepper with 1368 steps/turn

Using split-operator stepper with 1368 steps/turn

Using split-operator stepper with 1368 steps/turn

In [13]:

print "For t1 v6.6 lattice:" + str(lattice\_dict['t1\_1IO\_66']['lattice\_simulator'].get\_both\_tunes())

print "For t3 v6.6 lattice:" + str(lattice\_dict['t3\_1IO\_66']['lattice\_simulator'].get\_both\_tunes())

print "For t1 v8.2 lattice:" + str(lattice\_dict['t1\_1IO\_82']['lattice\_simulator'].get\_both\_tunes())

print "For t1 v8.2 lattice - detuned:" + str(lattice\_dict['t1\_1IO\_82\_dQ\_1']['lattice\_simulator'].get\_both\_tunes())

#print "For t1 v8.2 uncentered lattice:" + str(lattice\_dict['t1\_1IO\_82\_uncentered']['lattice\_simulator'].get\_both\_tunes())

print "For t3 v8.2 lattice:" + str(lattice\_dict['t3\_1IO\_82']['lattice\_simulator'].get\_both\_tunes())

For t1 v6.6 lattice:(0.300064734030028, 0.3000459461883418)

For t3 v6.6 lattice:(0.40264398946138624, 0.13435599909012663)

For t1 v8.2 lattice:(0.30007507878938977, 0.3000355421087709)

For t1 v8.2 lattice - detuned:(0.39844408797194025, 0.3987698808433909)

For t3 v8.2 lattice:(0.40265323055043917, 0.13434432538702376)

In [14]:

opts.save = False

opts.lattice\_name = 'IOTA 8-2 1IO standard lattice'

opts.lf\_fns = ['beta\_x','beta\_y','D\_x']

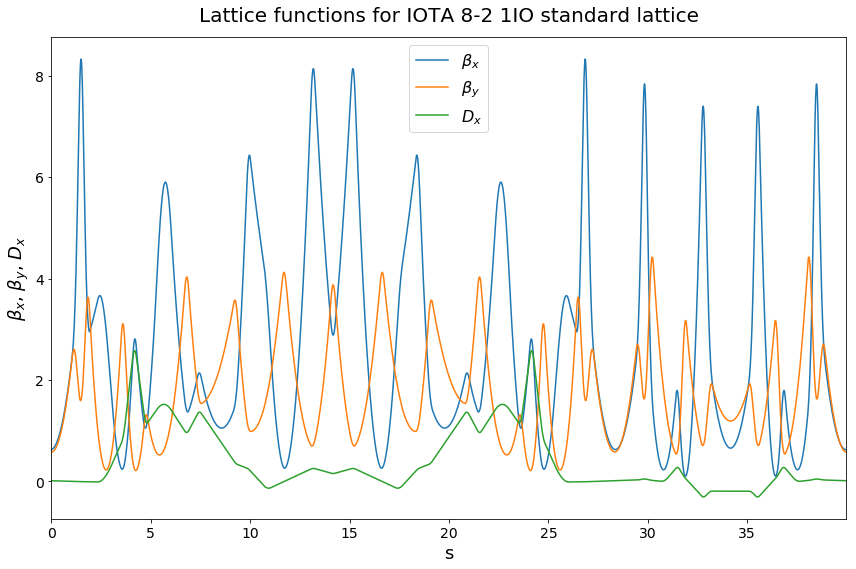
opts.lattice = lattice\_dict['t1\_1IO\_82\_dQ\_1']['lattice']

opts.lattice\_simulator = lattice\_dict['t1\_1IO\_82\_dQ\_1']['lattice\_simulator']

lfplot.plot\_sliced\_lattice\_functions(opts)

/home/vagrant/.pyenv/versions/py2/lib/python2.7/site-packages/matplotlib/figure.py:2267: UserWarning: This figure includes Axes that are not compatible with tight\_layout, so results might be incorrect.

warnings.warn("This figure includes Axes that are not compatible "

**Construct the beam and propagate it**

In [15]:

opts.lattice = lattice\_dict['t1\_1IO\_82\_dQ\_1']['lattice']

opts.lattice\_simulator = lattice\_dict['t1\_1IO\_82\_dQ\_1']['lattice\_simulator']

opts.stepper = lattice\_dict['t1\_1IO\_82\_dQ\_1']['stepper']

In [16]:

def dQ\_SC\_Gauss(N,emit,ref):

'''Return dQsc if given normalized emittance'''

r0 = 1.54e-18 #classical proton radius

bet = ref.get\_beta()

gam = ref.get\_gamma()

dQ = -1.\*r0\*N/(4\*np.pi\*emit\*bet\*gam\*gam)

#print gam\*gam\*gam

#print bet\*bet

return dQ

def dQ\_SC\_KV(N,emit,ref):

'''Return dQsc for a KV beam if given normalized emittance (2x that of Gaussian)'''

r0 = 1.54e-18 #classical proton radius

bet = ref.get\_beta()

gam = ref.get\_gamma()

dQ = -1.\*r0\*N/(2\*np.pi\*emit\*bet\*gam\*gam)

#print gam\*gam\*gam

#print bet\*bet

return dQ

g\_emit = 20.e-6 #TOTAL geometric emittance of IOTA beam

n\_emit = basic\_calcs.calc\_normalized\_emittance(g\_emit,opts.beta,opts.gamma)

current = 3.85\*1.e-3 #mA of current

l\_IOTA = 39.968229715800064 #length of lattice

rp\_perlength = current/(reference\_particle.get\_beta()\*scipy.constants.c\*scipy.constants.e)

n\_particles = rp\_perlength\*l\_IOTA

dQ = dQ\_SC\_Gauss(n\_particles,n\_emit,reference\_particle) #calc dQsc\_x

dQ\_KV = dQ\_SC\_KV(n\_particles,n\_emit,reference\_particle) #calc dQsc\_x

print "Assume a normalized total emittance of {} mm-mrad.".format(n\_emit\*1.e6)

print "At {} mA current, # of protons filling ring is {:e} or {:e} p+/cm.".format(current\*1.e3,n\_particles,n\_particles/(100.\*l\_IOTA))

print "Corresponding space charge tune shift in x is {} for KV distribution".format(dQ\_KV)

Assume a normalized total emittance of 1.46096398318 mm-mrad.

At 3.85 mA current, # of protons filling ring is 4.397346e+10 or 1.100210e+07 p+/cm.

Corresponding space charge tune shift in x is -0.100722585302 for KV distribution

**A few notes on distributions for clarity**

Note, for the kV-like distribution in coordinate space, the RMS value in x/y is 1/2 the maximum value. That is, in coordinate space, . For the Gaussian distribution in momentum space, the RMS value is simply the standard deviation (). The RMS emittance of the bunch (geometric) is then the product of these two values .

Thus, we have . So if we specify a maximum coordinate (e.g. ), then we can define

For a beam not at a waist, the beam should have a coordinate rotation applied by the twiss parameter matrix in each plane, but since we are injecting at beam waists we don't have to worry about this.

*Defining* . This follows from the convention of defining . For a KV-like distribution in coordinate space, we can relate the maximum x value to the RMS via . Thus, we have .

In [17]:

new\_vals = latticework.get\_starting\_lf(lattice\_dict['t1\_1IO\_82\_dQ\_1']['lattice\_simulator'])

Initial starting lattice functions: betax = 0.627584360895, betay = 0.579814047034, alphax = -0.034327819411, alphay = 0.0232141438299

In [20]:

n\_ppc = 10

opts.gridx = 32

opts.gridy = 32

opts.gridz = 1

n\_macro = n\_ppc\*opts.gridx\*opts.gridy

opts.macro\_particles = n\_macro

#======================= Now setup the bunch and other related options =====================

#Can compute theoretical lattice functions for perfect drift

opts.t = tval

opts.c = cval

opts.new\_tune = 0.3

opts.lnll = 1.8

vals = basic\_calcs.get\_base\_nll(opts.lnll, opts.new\_tune, opts.t, opts.c)

#Alternatively, we can account for asymmetries in the lattice by grabbing these values directly

new\_vals = latticework.get\_starting\_lf(lattice\_dict['t1\_1IO\_82\_dQ\_1']['lattice\_simulator'])

opts.betae = new\_vals[0]

opts.alphae = 0

opts.beta0 = new\_vals[0]

dpop = 0.0

opts.dpop = dpop #0.1% dpop

rp\_perlength = current/(opts.beta\*constants.c\*constants.e)

bunch\_length = opts.lattice.get\_length() #effective bunch length is iota lattice length

opts.real\_particles = rp\_perlength\*bunch\_length

opts.emit = 0.5\*g\_emit #Match using the geometric emittance

opts.emits = [opts.emit]

if myrank == 0:

#construct a bunch and make sure to add longitudinal momentum variation

#particles = elliptic\_beam6d.toyellipticalbeam6D(opts)

#particles = semi\_gaussian6d.semigaussianbeam6D(opts)

particles = standard\_beam6d.toyKVbeam6D(opts)

for index in range(len(opts.emits)):

bunch = particles[index]

#initialH,initialI = elliptic\_sp.calc\_bunch\_H(bunch,opts)

#bunch\_mean = np.mean(initialH)

#bunch\_std = np.std(initialH)

#bunch\_var = (bunch\_std/bunch\_mean)\*100

#print "Constructed bunch with {} macroparticles, having mean H: {} and std: {}%".format(opts.macro\_particles, bunch\_mean,bunch\_var)

#now add longitudinal momentum variation

#For random samples with mean = 0, sigma = sigma, use sigma\*np.random.randn(...)

#bunch[:,5] = opts.dpop\*np.random.randn(1,len(bunch))

bunch[:,4] = bunch\_length\*(np.random.random(len(bunch)) -0.5) #center at 0

bunch[:,5] = opts.dpop\*np.random.randn(1,len(bunch)) #set dp/p

np.savetxt('{}/my\_KV\_bunch\_82.txt'.format(opts.output\_dir),bunch) #write the bunch to a text file

bucket\_length = bunch\_length

particles\_file = '{}/my\_KV\_bunch\_82.txt'.format(opts.output\_dir)

myBunch = read\_bunch.read\_bunch(particles\_file, reference\_particle, opts.real\_particles, comm, bucket\_length)

# generated longitudinal coordinate is z position (beta\*c\*dt) but Synergia uses

# c\*dt. Divide by beta to get c\*dt.

local\_particles = myBunch.get\_local\_particles()

local\_particles[:,4] /= opts.beta

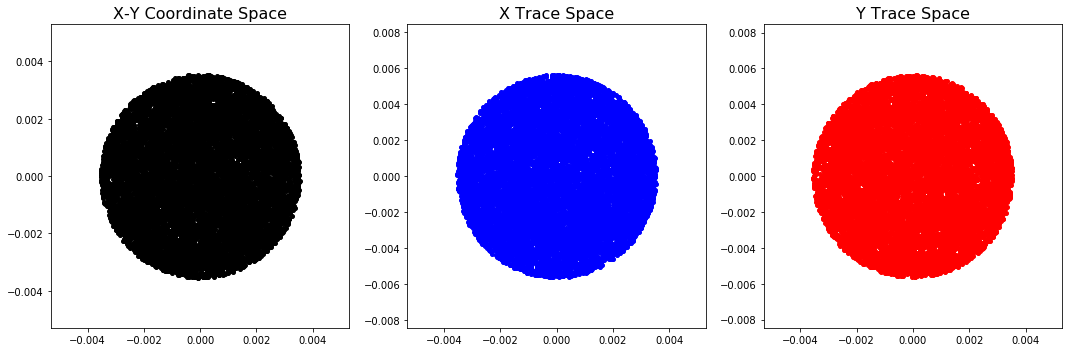
Initial starting lattice functions: betax = 0.627584360895, betay = 0.579814047034, alphax = -0.034327819411, alphay = 0.0232141438299

Read 10240 particles

In [21]:

pltbunch.plot\_bunch(myBunch)

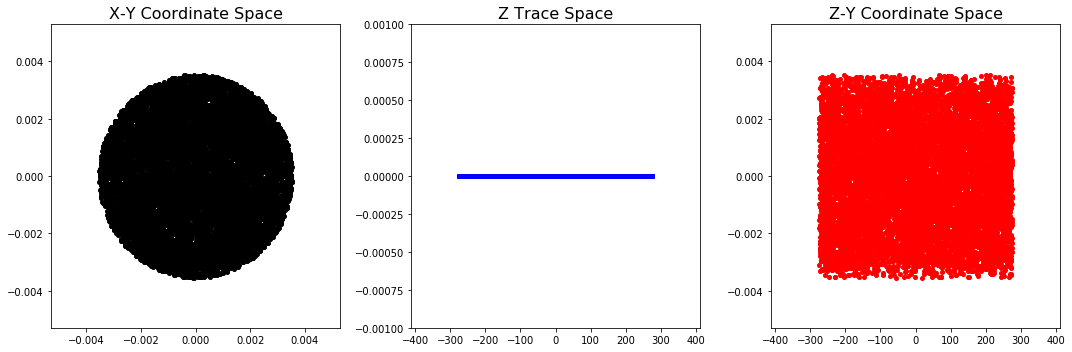
pltbunch.plot\_long(myBunch)

/home/vagrant/.pyenv/versions/py2/lib/python2.7/site-packages/matplotlib/axes/\_base.py:3443: UserWarning: Attempting to set identical bottom==top results

in singular transformations; automatically expanding.

bottom=0.0, top=-0.0

'bottom=%s, top=%s') % (bottom, top))

In [20]:

basic\_calcs.calc\_properties(myBunch,reference\_particle)

rms envelope x: 1.76283746063 mm

rms envelope y: 1.77591176371 mm

maximum x value is : 3.53243342517 mm

maximum y value is : 3.52823618638 mm

rms beta x: 0.621090115764

rms beta y: 0.631253498616

geometric emittance x: 5.00345414255 mm-mrad

geometric emittance y: 4.99619027759 mm-mrad

normalized emittance x: 0.365493314689 mm-mrad

normalized emittance y: 0.364962702434 mm-mrad

mean of xp^2 : 8.05592299018e-06

mean of yp^2 : 7.91471300918e-06

total geometric emittance x: 20.0906206467 mm-mrad

total geometric emittance y: 19.7202084649 mm-mrad

In [21]:

bunch\_simulator = synergia.simulation.Bunch\_simulator(myBunch)

#basic diagnostics - PER STEP

basicdiag = synergia.bunch.Diagnostics\_basic("basic.h5", opts.output\_dir)

bunch\_simulator.add\_per\_step(basicdiag)

#include full diagnostics

fulldiag = synergia.bunch.Diagnostics\_full2("full.h5", opts.output\_dir)

bunch\_simulator.add\_per\_turn(fulldiag)

#particle diagnostics - PER TURN

opts.turnsPerDiag = 1

particlediag = synergia.bunch.Diagnostics\_particles("particles.h5",0,0,opts.output\_dir)

bunch\_simulator.add\_per\_turn(particlediag, opts.turnsPerDiag)

In [22]:

opts.turns = 10

opts.checkpointperiod = 10

opts.maxturns = opts.turns+1

propagator = synergia.simulation.Propagator(opts.stepper)

propagator.set\_checkpoint\_period(opts.checkpointperiod)

propagator.propagate(bunch\_simulator,opts.turns, opts.maxturns,opts.verbosity)

workflow.cleanup(opts.output\_dir)

**Analysis**

**Analysis - Beam Envelopes**

In [23]:

opts.relpath = opts.output\_dir

opts.inputfile = opts.output\_dir + '/basic.h5'

opts.plots = ['x\_std', 'y\_std']

plotVals = diagplot.getPlotVals(opts.inputfile, opts.plots)

#define specific value arrays

xmaster = plotVals['s']

xstd = plotVals['x\_std']

ystd = plotVals['y\_std']

#we want to plot turn 0 and turn 10 on the same plot, so we should separate these specifically

interval = opts.steps

xstd\_0 = xstd[:interval]

xstd\_2 = xstd[1\*interval:2\*interval]

xstd\_end = xstd[-1\*interval:]

ystd\_0 = ystd[:interval]

ystd\_end = ystd[-1\*interval:]

#We can use the same s value for each plot

sval\_0 = xmaster[:interval]

In [24]:

fig = plt.figure(figsize=(8,6))

ax = plt.gca()

ax.plot(sval\_0,xstd\_0\*1.e3,'b-',alpha=0.7, label='Turn 1') #plot the 1st turn

ax.plot(sval\_0,xstd\_end\*1.e3,'g-',alpha=0.7, label='Turn %s' % opts.turns) #plot the 1st turn

axtitle = "Beam envelope evolution - $\sigma\_x$ over %s turns" % opts.turns

ax.set\_title(axtitle, y = 1.02, fontsize = 18)

ax.set\_xlabel("s [m]",fontsize=14)

ax.set\_ylabel("rms beam size $\sigma\_x$ [mm]",fontsize=14)

ax.tick\_params(axis='x', labelsize=14)

ax.tick\_params(axis='y', labelsize=14)

ax.set\_xlim([0,opts.lattice.get\_length()])

ax.legend()

sv\_title = 'beam\_envelope\_x\_compare\_1mA\_2D-600turns.pdf'

fig.tight\_layout()

#fig.savefig(sv\_title,bbox\_inches='tight')

In [25]:

fig = plt.figure(figsize=(8,6))

ax = plt.gca()

ax.plot(sval\_0,ystd\_0\*1.e3,'b-',alpha=0.7, label='Turn 1') #plot the 1st turn

ax.plot(sval\_0,ystd\_end\*1.e3,'g-',alpha=0.7, label='Turn %s' % opts.turns) #plot the 1st turn

axtitle = "Beam envelope evolution - $\sigma\_y$ over %s turns" % opts.turns

ax.set\_title(axtitle, y = 1.02, fontsize = 18)

ax.set\_xlabel("s [m]",fontsize=14)

ax.set\_ylabel("rms beam size $\sigma\_y$ [mm]",fontsize=14)

ax.tick\_params(axis='x', labelsize=14)

ax.tick\_params(axis='y', labelsize=14)

ax.set\_xlim([0,opts.lattice.get\_length()])

ax.legend()

sv\_title = 'beam\_envelope\_y\_compare\_1mA\_2D-600turns.pdf'

fig.tight\_layout()

#fig.savefig(sv\_title,bbox\_inches='tight')

**Analysis - Bunch Properties**

In [26]:

pltbunch.plot\_bunch(myBunch)

pltbunch.plot\_long(myBunch)

In [27]:

basic\_calcs.calc\_properties(myBunch,reference\_particle)

rms envelope x: 1.81644142818 mm

rms envelope y: 1.76716472325 mm

maximum x value is : 3.5669170034 mm

maximum y value is : 3.56399827214 mm

rms beta x: 0.655625613617

rms beta y: 0.620165276098

geometric emittance x: 5.03253593742 mm-mrad

geometric emittance y: 5.03554661874 mm-mrad

normalized emittance x: 0.367617687432 mm-mrad

normalized emittance y: 0.367837612281 mm-mrad

mean of xp^2 : 7.67592942207e-06

mean of yp^2 : 8.11968488534e-06

total geometric emittance x: 19.4057349879 mm-mrad

total geometric emittance y: 20.4817718331 mm-mrad

In [28]:

files = elliptic\_sp.get\_file\_list(opts)

twiss = twiss = elliptic\_sp.get\_toy\_twiss(opts)

rms\_table = []

for inputfile in files:

fn = inputfile[-7:-3]

parts = elliptic\_sp.get\_particles(inputfile)[1]

header= {}

header['s\_val'] = 0.

#norm\_coords = elliptic\_sp.normalized\_coordinates(header, part\_array, twiss)

x\_rms = basic\_calcs.get\_rms\_envelope('x',parts)

y\_rms = basic\_calcs.get\_rms\_envelope('y',parts)

rms\_table.append((int(fn),x\_rms,y\_rms))

#print "File {} : xrms = {:.2f} mm and yrms = {:.2f} mm".format(fn, x\_rms\*1.e3, y\_rms\*1.e3)

rms\_array = np.asarray(rms\_table)

In [29]:

num\_turns = rms\_array.shape[0]-1

fig = plt.figure(figsize = (8,6))

plt.subplot(1,1,1)

ax = plt.gca()

ax.plot(rms\_array[:,0],rms\_array[:,1], label = '$x\_{rms}$')

ax.plot(rms\_array[:,0],rms\_array[:,2], label = '$y\_{rms}$')

axtitle = "Beam envelope evolution over {} turns with KV beam".format(num\_turns)

ax.set\_title(axtitle, y = 1.02, fontsize = 18)

ax.set\_xlabel("Turn Number",fontsize=14)

ax.set\_ylabel("rms beam size [mm]",fontsize=14)

ax.tick\_params(axis='x', labelsize=14)

ax.tick\_params(axis='y', labelsize=14)

ax.set\_xlim([0,num\_turns])

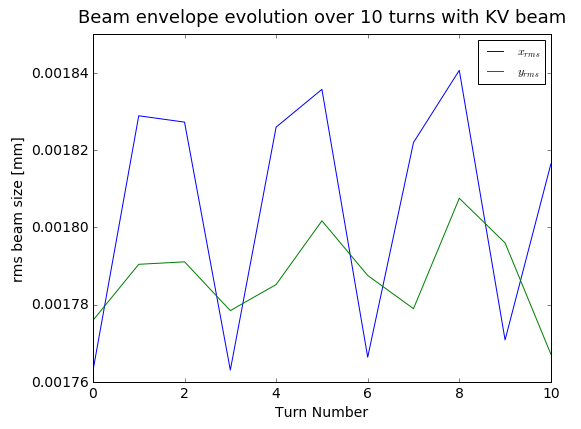
ax.legend()

sv\_title = 'Benchmark\_KV\_{}turns\_20mm-total-emittance.pdf'.format(num\_turns)

fig.tight\_layout()

fig.savefig(sv\_title,bbox\_inches='tight')

plt.show()



**Analysis - Tune depression**

In [30]:

def get\_particle\_coords(ID, num, opts):

'''Return particle (human) coordinates for particle with ID over first num turns'''

files = elliptic\_sp.get\_file\_list(opts)

twiss = elliptic\_sp.get\_toy\_twiss(opts)

part\_array = []

for index,outfile in enumerate(files[:num]):

#if index%20 == 0:

#print "Grabbing particle {} coordinates for file {}".format(ID,index)

particle = elliptic\_sp.get\_one\_particle(outfile,ID)

if index ==0:

part\_array = particle

else:

part\_array = np.vstack([part\_array,particle])

return part\_array

def estimate\_tune(x\_c, t\_s = 1):

'''Estimate the tune using an FFT of particle coordinates'''

num\_used = len(x\_c[t\_s:])

tv = np.arange(num\_used)\*1.0/num\_used

sp = np.fft.fft(x\_c[t\_s:])

#plt.plot(tv,sp.real)

smax = np.max(sp.real)

m\_ind = np.where(sp.real == smax)

Q\_guess =m\_ind[0][0]\*1./num\_used

if Q\_guess > 0.5:

Q\_calc = 1.- Q\_guess

else:

Q\_calc = Q\_guess

#print "Maximum is at {}".format(Q\_calc)

return Q\_calc

def estimate\_tune\_unwrap(x\_c,px\_c,t\_s):

'''Estimate the tune using a phase unwrap algorithm

Inputs:

-x\_c = normalized spatial coordinate

-px\_c = normalized momentum cooridnate

-t\_s = starting turn value from the array of coordinates

'''

ang\_norm = []

for x,y in zip(x\_c,px\_c):

if x > 0 and y > 0: #quandrant I

ang\_norm.append(np.arctan(y/x))

elif x < 0 and y > 0: #quandrant II

ang\_norm.append(0.5\*np.pi + (0.5\*np.pi - np.abs(np.arctan(y/x))))

elif x < 0 and y < 0: #quadrant III

ang\_norm.append(np.pi + np.abs(np.arctan(y/x)))

else: #quadrant IV

ang\_norm.append(1.5\*np.pi + (0.5\*np.pi - np.abs(np.arctan(y/x))))

#take diference between elements

diff\_ang = np.ediff1d(ang\_norm)

#adjust for wrapping by replacing positive values with 2pi-val

dff\_adjust = []

for val in diff\_ang:

if val > 0:

val = val - 2\*np.pi

dff\_adjust.append(val)

#now look at the last 50-80 turns

#np.mean(dff\_adjust[30:])

t\_s = 0

meantune = -1\*(np.mean(dff\_adjust[t\_s:]))/(2\*np.pi)

#print "{} is the mean tune for particle 0 after turn {}".format(meantune,t\_s)

return meantune

In [31]:

num\_t = opts.turns

IDlist = np.arange(200)

tune\_array = []

part\_array = []

twiss = twiss = elliptic\_sp.get\_toy\_twiss(opts)

header= {}

header['s\_val'] = 0.

files = elliptic\_sp.get\_file\_list(opts)[:num\_t]

for index,inputfile in enumerate(files):

#get all of the particles specifed by IDlist

p\_A = elliptic\_sp.get\_some\_particles(inputfile,IDlist)

norm\_coords = elliptic\_sp.normalized\_coordinates(header, p\_A, twiss)

if len(part\_array) == 0:

#if empty, then replace with norm\_coords - first turn

part\_array = norm\_coords

else:

part\_array = np.vstack((part\_array,norm\_coords))

#reshape array to be indexable by ID number

new\_PA = part\_array.reshape(num\_t,len(IDlist),4)

x\_tune\_array = []

y\_tune\_array = []

for ID in IDlist:

x\_tune\_array.append(estimate\_tune(new\_PA[:,ID,0]))

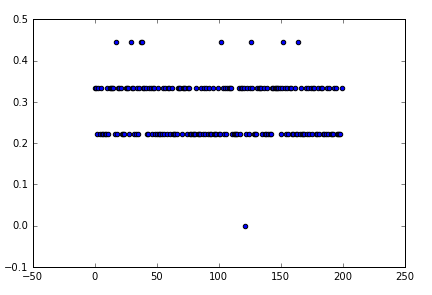
y\_tune\_array.append(estimate\_tune(new\_PA[:,ID,2]))

In [32]:

plt.scatter(IDlist,y\_tune\_array)

Out[32]:

<matplotlib.collections.PathCollection at 0x7f012a19d9d0>

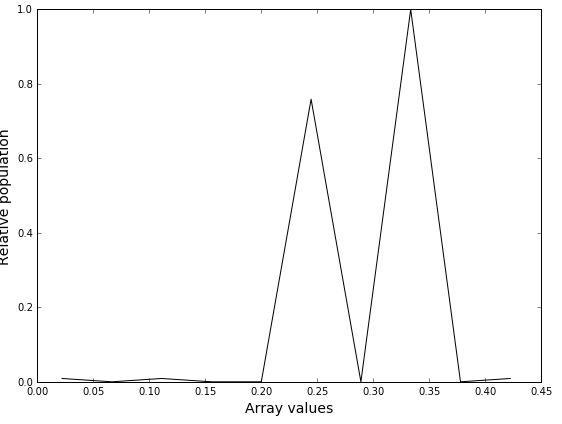


The tune spread is easier to observe using a histogram/distribution plot. Note that because we only examined a small number of turns, our sampling is rather low, and as a result our resolution for something like a histogram is pretty bad.

In [33]:

utils.plot\_distribution(x\_tune\_array,10)

Out[33]:



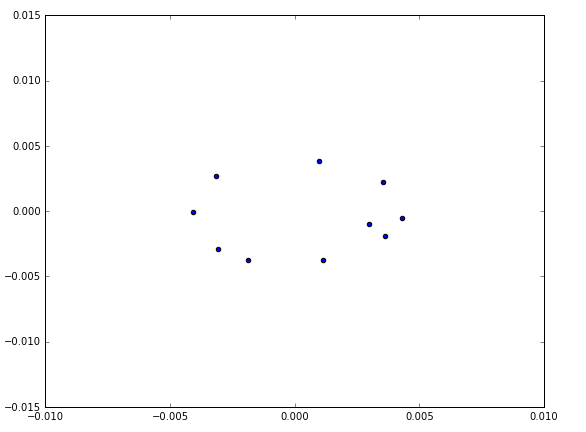
We can similarly examine a single particle's trajectory in its normal form.

In [34]:

plt.scatter(new\_PA[:50,0,0], new\_PA[:50,0,1])

Out[34]:

<matplotlib.collections.PathCollection at 0x7f0120161350>



In [ ]: