Simulating a Particle Storage Ring

060196497

May 27, 2009

PHY342

1 Simulation

1.1 Description

The physical system being simulated is that of a magnetic storage ring for particles, such as the Stanford Positron Electron Accelerating Ring (SPEAR). In a storage ring there is a toroidal cavity containing a strong vacuum, surrounded by a series of magnets (dipole, quadrupole, etc.), or more atypically electric poles. These create an electromagnetic field inside the ring, which is used to influence the path of charged particles in order to keep them orbiting in a narrow ring. The particles are non-interacting and their spin is neglected, as modelling these properties has been shown to be of little utility over non-interacting particles, whilst introducing more complexity and overhead according to Banford [1]. The full derivation of the mechanics is given in Appendix A, producing matrices to transform each particle's position and divergence between one infinitesimal section of the ring to the next.

Since machines which are capable of infinite computation in a finite time are still theoretical [2], the true results of integrating around the ring cannot be obtained numerically, thus the infinitesimal size dz is replaced with a small, but finite, size, giving approximate results.

The rings being simulated are given fixed properties at creation, such as field index, circumference/radius and beam width and height, as well as properties for their simulation such as the number of sections to divide the ring into (and hence the size of dz), the number of particles to contain and the number of revolutions to perform. The particles are created using a Monte Carlo technique, giving a uniform distribution across the width ($\pm x_{max}$) and height ($\pm y_{max}$) of the beam, as well as uniformly distributed divergences between the maximum/minimum divergences radially and vertically ($\pm x'_{max}$ and $\pm y'_{max}$), defined as $\frac{x_{max}}{r_0}$ and $\frac{y_{max}}{r_0}$ respectively (where r_0 is the radius from the centre of the ring to the centre of the beam). These are then inserted into the ring at an arbitrary section and the simulation is begun.

Whilst these values only appear once in the code, and are thus easily reconfigurable, in order to reduce the number of variables being considered the storage rings simulated have a beam width and height of 0.06m ($x_{max} = y_{max} = 0.03$) and a circumference of 2m ($\therefore\ r_0 = \frac{2}{2\pi} = 0.318$). The number of particles, number of revolutions, number of sections and field index were varied.

1.2 Results

1.2.1 Field Index

The first property investigated was the variation in acceptance (equal to the fraction of remaining particles) as the field index n was varied from $0 < n < 1$ for 20 rings of 40 sections each, containing 10,000 particles making 50 revolutions. Figure 1 shows the results of these simulations.

The shape obtained is a combination of the x and y acceptance, shown in figure 2 (slight deviations between the green line of $1-(x\,acceptance + y\,acceptance)$

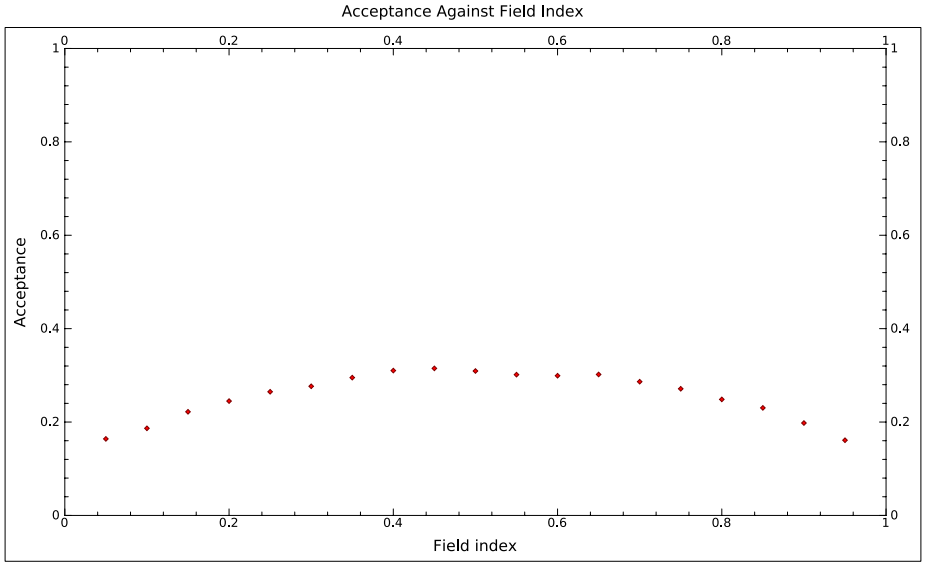

Figure 1: Acceptance against field index (10,000 particles)

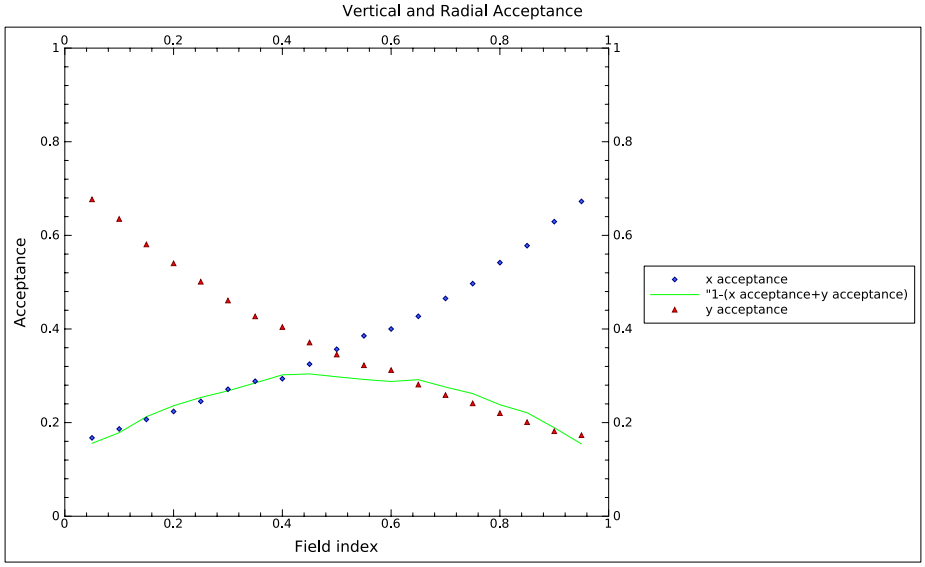

Figure 2: Vertical and radial acceptance components (10,000 particles)

and the total acceptance from figure 1 are due to particles which escape in both the x and y directions within a single step).

The symmetric shape of the total acceptance and the complementary shapes of the x and y acceptances can be explained by considering the transformation matrix which produced them (see Appendix A for its derivation)

$$\begin{bmatrix} \cos\left(\sqrt{1-n}\frac{dz}{r_0}\right) & \frac{r_0}{\sqrt{1-n}}\sin\left(\sqrt{1-n}\frac{dz}{r_0}\right) & 0 & 0 \\ -\frac{\sqrt{1-n}}{r_0}\sin\left(\sqrt{1-n}\frac{dz}{r_0}\right) & \cos\left(\sqrt{1-n}\frac{dz}{r_0}\right) & 0 & 0 \\ 0 & 0 & \cos\left(\sqrt{n}\frac{dz}{r_0}\right) & \frac{r_0}{\sqrt{n}}\sin\left(\sqrt{n}\frac{dz}{r_0}\right) \\ 0 & 0 & -\frac{\sqrt{n}}{r_0}\sin\left(\sqrt{n}\frac{dz}{r_0}\right) & \cos\left(\sqrt{n}\frac{dz}{r_0}\right) \end{bmatrix}$$

Here the dependency on the field index is clear. Taking $\alpha \equiv 1-n$ and $\beta \equiv n$ this becomes

$$\begin{bmatrix} \cos\left(\sqrt{\alpha}\frac{dz}{r_0}\right) & \frac{r_0}{\sqrt{\alpha}}\sin\left(\sqrt{\alpha}\frac{dz}{r_0}\right) & 0 & 0 \\ -\frac{\sqrt{\alpha}}{r_0}\sin\left(\sqrt{\alpha}\frac{dz}{r_0}\right) & \cos\left(\sqrt{\alpha}\frac{dz}{r_0}\right) & 0 & 0 \\ 0 & 0 & \cos\left(\sqrt{\beta}\frac{dz}{r_0}\right) & \frac{r_0}{\sqrt{\beta}}\sin\left(\sqrt{\beta}\frac{dz}{r_0}\right) \\ 0 & 0 & -\frac{\sqrt{\beta}}{r_0}\sin\left(\sqrt{\beta}\frac{dz}{r_0}\right) & \cos\left(\sqrt{\beta}\frac{dz}{r_0}\right) \end{bmatrix}$$

With this notation the similarity of the radial transformation (top left) and the vertical transformation (bottom right) is clear as α and β appear in exactly the same form. In the interval taken, $0 < n < 1$, the values of α and β are reflections of each other about $n = 0.5$.

1.2.2 Particle Number

The top graph of figure 3 superimposes similar results to figure 1, but for simulations of 10, 100, 1000 and 10000 particles, showing a convergence as the effect of each particle on the result is reduced. This convergence is shown more quantitively in the lower graph, which only takes the acceptance when $n = 0.5$, and extends the final acceptance value of 0.307 (for 11,000 particles) across the graph for comparison. This result is to be expected, since the particles are non-interacting and thus the only difference between the simulations is the average approaching the expected result (known as the "Law of large numbers" [3]).

1.2.3 Rotations and Steps Taken

Next to be considered is the number of rotations the particles make around the ring, and whether travelling further reduces the number of particles remaining. The effect can be seen in figure 4, where no dependency is observed. This is due to the simple harmonic motion which the particles follow (see Appendix A), causing those in unmaintainable orbits to escape in their first oscillation as they reach maximum displacement. Those remaining are oscillating within the

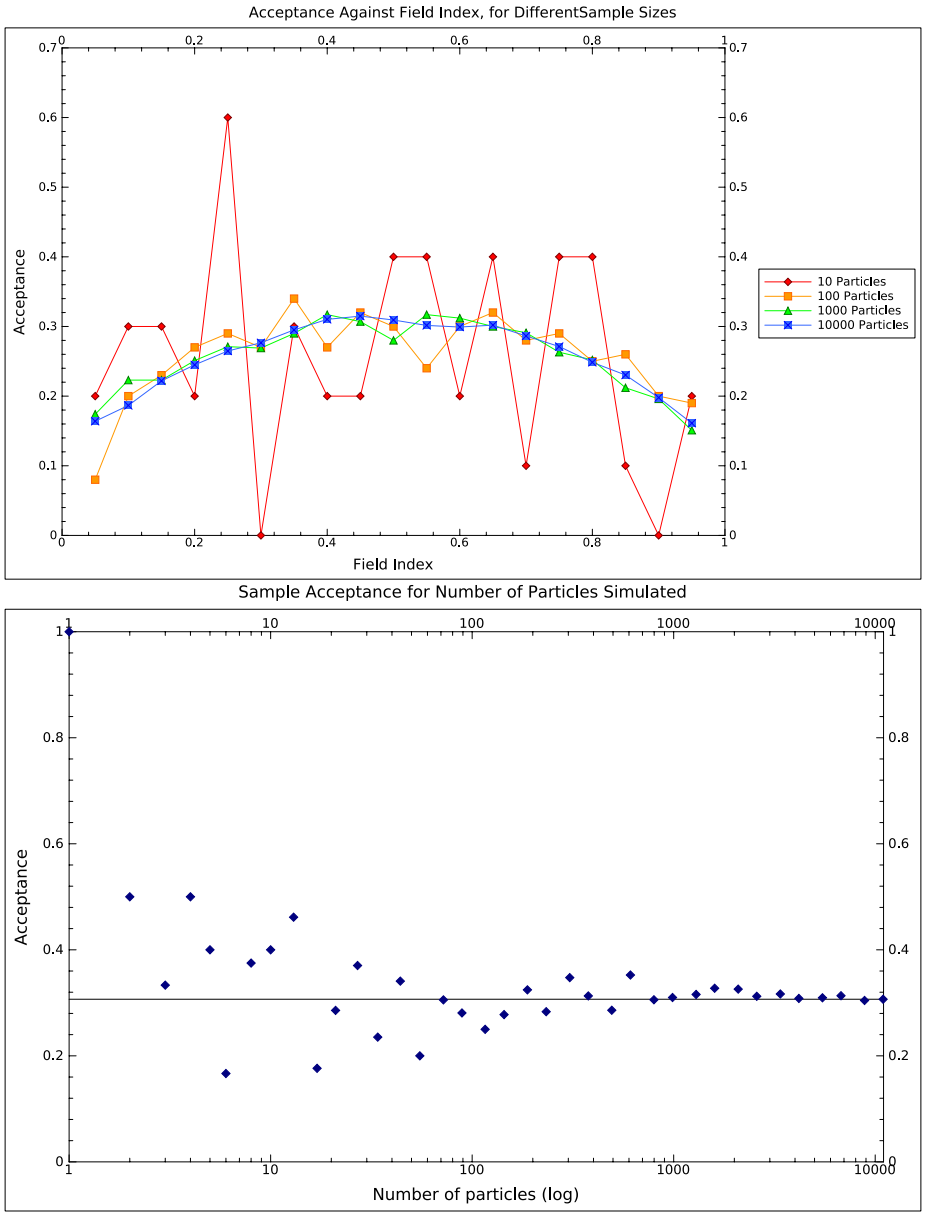

Figure 3: Acceptance converging for larger particle numbers

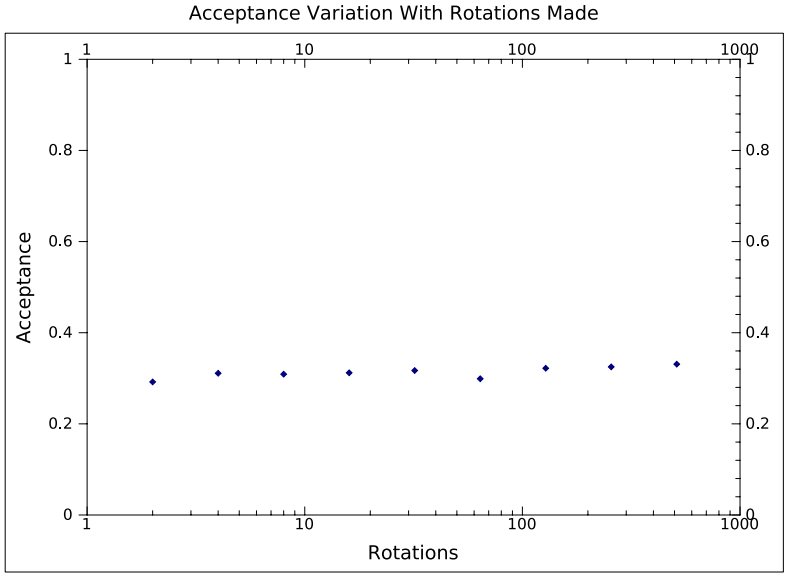

Figure 4: Acceptance variation with rotations made

confines of the beam, and thus can continue to do so for a prolonged period. A closer inspection of particle losses as they orbit their rings is given in figure 5, where the individual sections are considered rather than full rotations (which occur at every vertical line). The majority of particles which escape do so within the first rotation, whilst figure 6 shows the path of a particle traced from the start of the simulation until it escaped the ring, confirming that the simple harmonic behaviour is the cause. In the plot "particle 1" oscillates well within the beam width and height, whilst "particle 2" exits the beam and escapes after traversing 14 sections.

2 Implementation

2.1 Platform choice

To implement the simulation requires an assessment of development options. Whilst all notable programming languages are Turing complete, and thus equivalent, this does not imply that a direct translation between all languages is possible. Where it is not, equivalence can only be achieved by emulation, writing a Turing machine within the target language which accepts as input the source language. This technique of emulation allows ever more sophisticated Turing machines to be constructed, capable of accepting more complex and structured input.

In hardware this allows different silicon circuits to emulate standard hardware architectures, such as *ARM* and *POWER*, whilst in software it allows

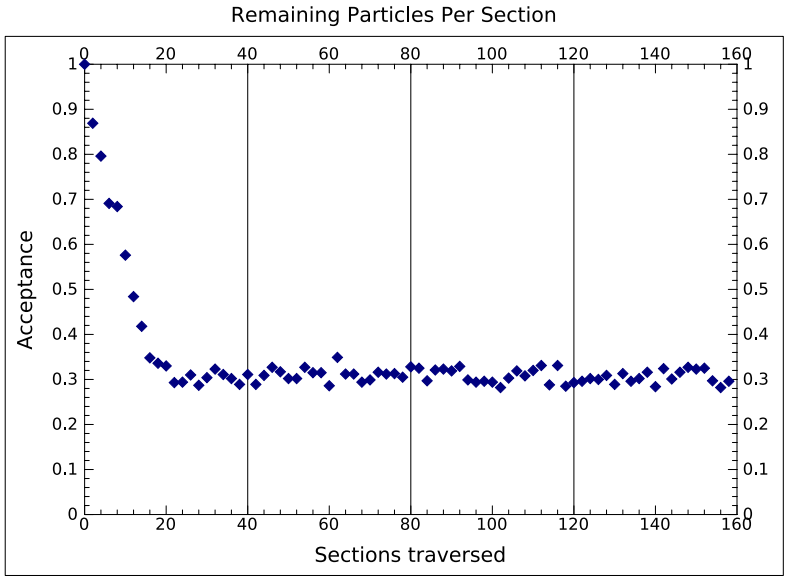

Figure 5: Remaining particles per section (40 sections per rotation)

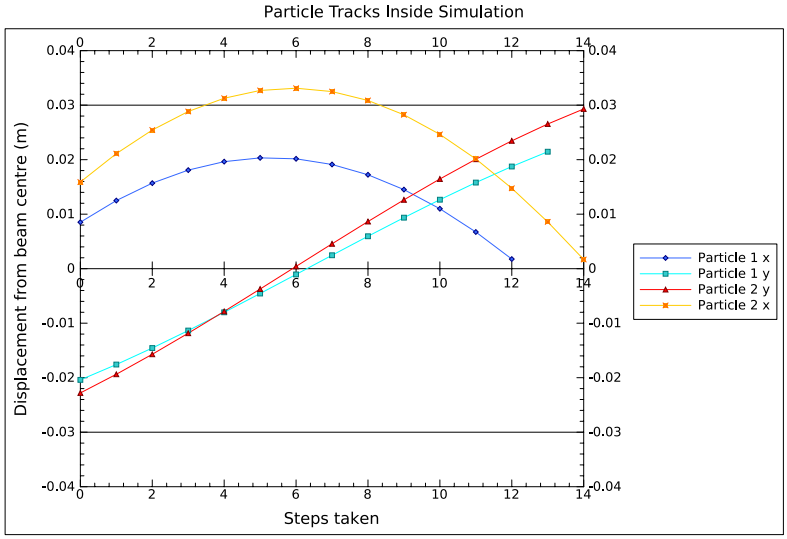

Figure 6: x and y paths for two sample particles (horizontal lines are $\pm x_{max}$ and $\pm y_{max}$)

implementation of higher-level languages such as C and FORTRAN by generating an equivalent, yet much less comprehensible, machine code emulation of the more abstract code. More abstract languages, such as Java, Ruby and Python, can be built by emulating an appropriate machine in lower level language like C (eg. *CPython*) or machine code (eg. a compiler generated by *PyPy*).

These "high level" languages, as opposed to the "low level" languages which are more similar to those of computer circuitry, are more flexible by definition (low level languages are a subset of high level languages), and are thus more appropriate for representing a wide range of algorithms since any data structures, control flow, behaviour and syntax can be created specifically to suit the problem being addressed. For example, languages which allow operator overloading can use regular multiplication syntax (for example a=b*c) to represent other operations such as matrix multiplication, and language support for linked lists and tail recursion can often reduce boilerplate significantly. The disadvantage of high level languages is the performance cost paid by the emulation. Each level of abstraction implements a Turing machine, translating one instruction set into a simpler one in a general way, thus growing the number of operations at each stage and losing efficiency due to the generality of the translation. For example a terse Python program will generate a large amount of CPython bytecode, which in turn will call an even larger number of C instructions, which sends an even larger number of machine instructions. Similarly a program written in POWER machine code will not perform as well in a POWER emulator (for example PearPC) as opposed to a real POWER machine (for example Microsoft's XBox 360).

Considering the advantages and disadvantages of high level languages, it was decided that the clarity, modularity and compact nature inherent to high level code, making its correctness simpler to verify, outweighed the performance losses associated. The possible candidates decided upon were C, C++, Java, FORTRAN, Python, Haskell and Erlang. The merits of each are discussed below.

2.1.1 C

C is a statically typed, structured programming language which is highly portable (for example, the GNU Compiler Collection alone supports over 50 machine architectures) and has been the subject of optimisation experiments for decades. Although it has this pedigree, as well as an extensive collection of libraries, examples and a large community following, C has numerous disadvantages which made it undesirable for this project.

Firstly the sophistication of data structures afforded to a C program is severely limited. C contains a few arbitrary numerical types, such as "int", "float" and "char", along the lines of the *IEEE 754* standard, which are built in to the language implementation and unchangable. Adding types is achieved by enumerating values for that type, automatically assigning each possible value an integer number (which can cause headaches later as they are indistinguishable from non-enumerating integers). This inflexibility extends to C's type system,

which assigns a type to each variable given in the code, forcing the declaration of that type by the developer. New data structures must be constructed through the use of "structs", unordered mappings of key/value pairs, and through fixed-length, typed arrays. More complex typing systems can be used (eg. GObject), but these are essentially emulations of C++.

The typing of each variable comes from the minimalist memory model of C, where memory is treated as an array of addressable bytes. Each variable has a pointer specifying the memory location at which its storage begins, and a type which allows determination of the size (and thus end) of the storage. Memory banging in this way is prevalent throughout C, often being the only way to achieve what other languages have built in. For example only a single variable of a built-in type can be returned from a C function, thus for anything more complex pointers are used to find the location and the memory is altered globally. With these severe restrictions, and the hacks and boilerplate required to work around them, C is not the best candidate for this simulation.

2.1.2 C++

C++ is very similar to C, although it has an improved type system built in. This type system is a primitive form of Object Oriented programming, with objects being wrappers around collections of the built in types, other objects and functions, just like a struct. The difference comes through the class system of C++, which groups the behaviour of objects in to classes approaching the level of built in types, but still just remaining a wrapper.

Whilst better than C for this project, C++ still suffers from most of C's restrictions and requires even more boilerplate to implement an algorithm. For this reason C++ was discarded.

2.1.3 Java

Java is syntactically similar to C++, but abstracts away some of the hacks remaining from C. For example, in Java the distinction between a pointer and a value is based on context, rather than them being explicitly differentiated from each other. As such Java's memory model is only accessible to Java programs through Java code, there is no rewriting of memory addresses by the program.

Whilst Java takes the class-based object system of C++ slightly further, treating objects in almost the same way as its built in, non-object types, it is still simply a wrapper around the built in types, which are not changable.

Java is clearer than C and C++ code due to the lack of pointers and the extended use of objects, and its inclusion of a garbage collector reduces some code overhead, however despite this the boilerplate for Java can be even higher than that of C++, and the enforcement of Java's style makes algorithm implementation difficult unless the algorithm happens to exactly fit Java's restrictive definition of correct.

This makes Java unsuitable for this project.

2.1.4 FORTRAN

FORTRAN is a heavily numerical language, stateful and fast, which is a well suited paradigm to a Physics simulation project, however code organisation and data structures are rather limited. This makes it useful, but not ultimately the implementation language.

2.1.5 Python

Python is a dynamic, late-bound language which does not suffer from many of the static restrictions of C, C++ and Java since it is entirely object oriented. In Python there are no base types, since everything is an object. Whilst the class-based object model used in C++ and Java is also used in Python, the requirement to give variables explicit types is eliminated by making every variable a pointer. Since pointers are always integers this removes any distinction between them, even though they can point at any Object in memory. Python is therefore "duck typed", following the adage "If it walks like a duck and quacks like a duck, I would call it a duck". This is in reference to accessing a member of an object , which Python will always attempt regardless of whether the object contains such a member or not, raising an Exception (which can be caught) in the latter case. Duck typing in combination with the "Easier to ask forgiveness than permission" method of exception handling reduces a lot of boilerplate code, leaving in place highly readable code. The extra polymorphism introduced also reduces code duplication and the requirement for a pervasive naming scheme (such as Hungarian notation). The pervasiveness of Python's object model extends to classes and functions, which are first class language constructs. Late binding allows classes to be modified on the fly and high level functions (those which act on other functions) to be constructed in a general way.

Another advantage to Python is the resemblance of its syntax to that of pseudocode, making the implementation of a pseudocode algorithm often a matter of merely tweaking the syntax, such as adding a colon after "if" and "for" lines. The syntax also has built in support for lists (mutable object sequences of mutable length) by simply writing "[element_1, element_2, ... element_n] and dictionaries (mutable associative arrays, mapping key objects to value objects) by writing "{key_1:value_1, key_2:value_2, ... key_n:value_n}. These make Python code very readable, especially when used in loops of the form "for element in list:" and "for element in dictionary.keys():".

Python's implicit, stateful paradigm is well suited to the simulation being approached. Its polymorphism, achieved through duck typing, allows modularity without loss of terseness.

2.1.6 Haskell

Haskell is a lazy, stateless, functional programming language. Being functional and stateless, Haskell is well suited to evaluating Mathematics, since Mathematics is inherently concurrent, and thus the concept of reassigning variables (outside of specified "let" statements) makes no sense.

Haskell is lazy in that its functions are evaluated when required to produce output, but other than that they are simply 'remembered' as "thunks". This allows convenient constructions such as infinitely long lists and "Currying", supplying functions with fewer arguments than they require in order to produce new functions which no longer require those arguments to be supplied. This fits elegantly into Haskell's type system, which allows extremely general, verifiably correct functions to be constructed.

However, since variables are immutable in Haskell this comes with the cost of a much higher memory footprint (since new variables must be constructed to store new values, rather than replacing old ones). For a simulation, which only depends on the current state and the functions which transform it into the next state, the overhead of remembering every state of every element of the system at every point in time is too much of a cost. This could be overcome through the use of thunks, but would make removing escaped particles difficult.

2.1.7 Erlang

Erlang is a distributed, functional, actor based programming language designed to spread computation transparently over thousands of machines or more. Whilst this is in itself a killer feature for the language, it is incredibly difficult to verify the correctness of the concurrent code. Though sophisticated testing infrastructure exists, often the majority of an Erlang application is tests. For a scientific project this level of uncertainty is not acceptable.

2.1.8 Decision

Python was chosen as the most appropriate language to use, since it allows the expression of the relevant algorithms in a concise and understandable manner. Whilst typical Python implementations do not perform as well as lower level languages such as C or machine code, this is a temporary issue which can be resolved through more sophisticated implementations. Donald Knuth famously said "Premature optimization is the root of all evil", and thus the best optimisation should be performed at run time, as the confusion it creates in the thought processes before this point works to the detriment of a project. Some attempts at optimisation are discussed below.

2.2 Implementations

2.2.1 First generation: x and y matrices

Once the formulas in Appendix A had been derived, implementing a correct simulation in Python took around 30 minutes. The approach taken was to split the problem into the classes Particle (representing a particle), Ring (representing a ring) and Headless (representing a non-interactive simulation). Each Particle object contains two matrices, the $\begin{bmatrix} x \\ x' \end{bmatrix}$ matrix and the $\begin{bmatrix} y \\ y' \end{bmatrix}$ matrix. The

implementation of these matrices is taken from the *Numpy* library of numerical and scientific Python code. The Ring contains a list of particles and two matrices, the $\begin{bmatrix} \cos\left(\sqrt{1-n}\frac{dz}{r_0}\right) & \frac{r_0}{\sqrt{1-n}}\sin\left(\sqrt{1-n}\frac{dz}{r_0}\right) \\ -\frac{\sqrt{1-n}}{r_0}\sin\left(\sqrt{1-n}\frac{dz}{r_0}\right) & \cos\left(\sqrt{1-n}\frac{dz}{r_0}\right) \end{bmatrix}$ matrix for x transformations and the $\begin{bmatrix} \cos\left(\sqrt{n}\frac{dz}{r_0}\right) & \frac{r_0}{\sqrt{n}}\sin\left(\sqrt{n}\frac{dz}{r_0}\right) \\ -\frac{\sqrt{n}}{r_0}\sin\left(\sqrt{n}\frac{dz}{r_0}\right) & \cos\left(\sqrt{n}\frac{dz}{r_0}\right) \end{bmatrix}$ matrix for y transformations.

The Ring can be told to step its list of Particles through a number of sections (ie. apply the matrices the given number of times). At each step the Particles list is pruned of those which fall outside the Ring's boundaries. The full simulation is achieved by performing $N_{sections} \times N_{rotations}$ steps.

This works well, but is rather slow. Timing results show a total running time of 140 seconds to simulate 19 Rings (with field indices from 0.05-0.95 inclusive), with 40 sections each, with the simulation sending 100 particles around each 50 times. This is a total of 7.6 million multiplications of 2x1 vectors by 2x2 matrices, around 50,000 multiplications per second.

Some speed increases were achieved through use of Python's functional abstractions, replacing the loop multiplying the matrices with a call to *map* the appropriate function to the particles list, and the loop checking for escaped particles was replaced with a call to *filter* the list using a position-checking function. This made the code slightly clearer and brought the running time for the setup described above to 55 seconds, a reduction of 60%.

Further optimisation focused on making the simulation run concurrently across both CPU cores of the machine being used for development. Use of the *Threading* module was abandoned due to the CPython implementation's "Global Interpreter Lock" which prevents multiple threads from running concurrently in one process. Next the *Processing* module was used, which has an identical API to *Threading* but spawns multiple processes rather than threads. These run concurrently, giving a 100% speed increase on a dual-core machine, however the lack of shared memory between processes made communicating the results into the output stage difficult. The *Multiprocessing* module was then used, which once again follows the same API, but offers queues sharable between processes, which made it more straightforward to send results back, and even allowed regular progress to be reported, leading to a rudimentary GUI written using the *Pygame* graphics library.

2.2.2 Second generation: Combined x and y matrices

The next iteration of the simulation forked it and replaced the radial and vertical vectors with a single, 4 row vector, and the associated transformation matrices with a single 4x4 matrix, shown at the end of Appendix A. The main advantage to this is that the Particle class becomes simply a wrapper around its Numpy matrix, however this iteration never achieved satisfactory results.

2.2.3 Third Generation: FORTRAN enhanced

Using the *f2py* tool it is trivial to call FORTRAN subroutines from Python, thus the simulation was forked and FORTRAN code was incorporated for quickly creating the large numbers of particles required, stepping through the sections and multiplying matrices stored in row-major order (ie. row1 column1, row1 column2, row2 column1, row2 column2, etc.). The use of FORTRAN, however, gave consistently wrong results, with each even-indexed Ring losing all its Particles whilst the odd-indexed Rings gave the results expected. This occurred for the custom FORTRAN as well as simple calls to FORTRAN's built-in MATMUL subroutine to multiply two matrices. After much debugging the embedded FORTRAN was abandoned.

2.2.4 Fourth Generation: C++ Compiled

The *shedskin* tool allows compilation of a subset of Python, which does not utilise its dynamic abilities, into C++ code. This can then be compiled and used by any other Python program as if it were native, although losing the duck typing of its functions. The simulation was forked and hacked until it did not use any dynamic types or other dynamic features, then was compiled to C++ by shedskin and then to machine code with GCC. After much further hacking the compiled modules could be used concurrently by each process. This compiled version, however, did not give satisfactory results, thus after much debugging it too was abandoned.

2.2.5 Fifth Generation: Rewritten

The final iteration of the simulation was made from scratch, once work on the previous solutions ensured that the problem was better understood. This version is pure Python, consisting of a Matrix class (implementing a custom Python implementation of general matrices, including operator overloading) a Particle class (containing a Matrix and whether the particle has escaped vertically or horizontally), a RingSection class (containing a Matrix and multiplies with a Particle), a Ring class (containing a list of RingSections, a list of Particles and an index of the current position in the sections list) and a Simulation class (which initialises everything). The source for this implementation is in Appendix B. Various forks of this code were used to generate the statistics in section 1.2.

Some optimisations were attempted using Pyrex to compile Python into C but this produced some speed ups and some slow downs. The code was not modified. Attempts were also made to JIT compile with PyPy, but the source would not build for lack of memory.

2.3 Evaluation

In total for this simulation there have been 5 Python implementations, 3 FORTRAN subroutines, 2 Haskell attempts, 1 Reia (Erlang) attempt and 2 attempts to construct C extensions for Python. In the end a pure Python approach was

taken, as its simplicity cannot be made up for by performance optimisations. The simulation works, and will continue to work for as long as a computer is capable of running Python 2.x scripts. The code has been tested in 3 different Python implementations, *CPython 2.5*, *Unladened Swallow* and *PyPy 1.0*. Whilst speed could improve dramatically, the simulation is not the correct place to fix this, the correct area to address speed issues is in the interpreter/compiler. There are on-going efforts to improve Python performance such as the *StarKiller* and *shedskin* C++ translators, the *Pyrex* C translator, the *Psyco* x86 machine code translator and its successor the *PyPy* project, which JIT compiles to CLI (Common Language Infrastructure) and Java bytecode, LLVM instructions, C and machine code (amongst other things). This is an active area of research, and the widespread use of Python ensures it gets some attention (for instance Google's *Unladened Swallow* project is applying as much state-of-the-art knowledge to CPython as possible)

3 Appendices

Appendix A: Derivation of Particle Mechanics

Here we derive the equations of motion for a charged particle in a magnetic storage ring from classical electromagnetism, representing them in matrix form.

Firstly it is known from Maxwell's equations that

$$\nabla \cdot \mathbf{B} = 0 \tag{1}$$

where $\mathbf{B}$ is a magnetic field, from electromagnetism that the force on a particle with charge e, moving through a magnetic field $\mathbf{B}$ with a velocity v perpendicular to the field is given by

$$\mathbf{F}_{\text{MAGNETIC}} = e\mathbf{v} \times \mathbf{B} \tag{2}$$

from classical mechanics the centripetal force on a body with mass m in a stable orbit of radius r is given by

$$\mathbf{F}_{CP} = \frac{mv^2}{r}$$

and from Newton the net force on a body with mass m and displacement $\mathbf{d}$ is given by

$$\mathbf{F}_{NET} = -m\ddot{\mathbf{d}}$$

Considering an infinitesimal section of the ring there is no curvature, so Cartesian coordinates can be used without complication. For simplicity the z axis is defined as the propagation direction, making the length of the section dz. This gives equation 1 the form

$$\frac{\partial B_x}{\partial x} + \frac{\partial B_y}{\partial y} + \frac{\partial B_z}{\partial z} = 0$$

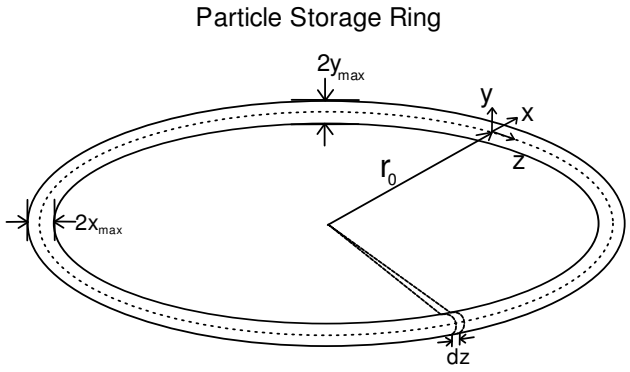

Figure 7: Storage ring diagram. Origin is at the centre of the ring.

Since the ring is symmetric, and thus has an arbitrary origin, it must be true that

$$\frac{\partial B_z}{\partial z} = 0$$

$$\therefore \frac{\partial B_x}{\partial x} + \frac{\partial B_y}{\partial y} = 0$$

The coordinates can be further defined such that the x axis is always parallel to the major radius of the ring, r_0, with its origin at the centre of the ring, and the y axis is perpendicular to the x and y axes with it origin on the median plane containing the ring. Motion parallel and perpendicular to the ring's major radius (hereafter referred to as radial and vertical respectively) can be considered independently.

3.1 Radial Motion

The radial component of motion is parallel to the x axis. Considering only this direction, equation2 is simply

$$F_x = evB_y$$

where v is the component of the velocity perpendicular to the y axis.

By equating the radial forces, and taking x to be the radial displacement from the particle beam's centre, the equilibrium radius can be found

$$evB_y = \frac{mv^2}{r_0 + x} \tag{3}$$

$$\frac{evB_y}{mv^2} = \frac{1}{r_0 + x}$$

Since $x \ll r_0$, the first two terms of the binomial expansion of the right hand side can be used as a valid approximation

$$(r_0 + x)^{-1} \simeq \frac{1}{r_0} - \frac{x}{r_0^2} = \frac{r_0 - x}{r_0^2}$$

$$\therefore \frac{evB_y}{mv^2} \simeq \frac{r_0 - x}{r_0^2}$$

$$evB_y \simeq \frac{r_0 - x}{r_0^2} mv^2$$

This describes the forces in the z direction, but to generalise to the plane perpendicular to y the radial forces need to be considered. By including these, in the form of $m\ddot{x}$, all forces in the plane of the ring can be related by equating the required centripetal force with the sum of the electromagnetic and the radial force

$$m\ddot{x} + e\mathbf{v}B_y = \frac{r_0 - x}{r_0^2} m\mathbf{v}^2$$

$$e\mathbf{v}B_y = m\left(\frac{r_0 - x}{r_0^2}\mathbf{v}^2 - \ddot{x}\right) \tag{4}$$

Since the magnetic field $\mathbf{B}$ changes through space, it is necessary to consider B_y as a function of x. Since particle positions are characterised by the radius r_0 and their displacement from it x, where $x \ll r_0$, the vertical field component at a particle's position can be approximated using Euler's method

$$B_y(r_0 + x) \simeq B_y(r_0) + x\frac{\partial B_y}{\partial x}$$

Now the field index n can be defined such that

$$n \equiv -\frac{r_0}{B_y} \cdot \frac{\partial B_y}{\partial x}$$

to give the vertical field component the following form

$$B_y(r_0 + x) \simeq B_y(r_0) + x\frac{\partial B_y}{\partial x}$$

$$B_y(r_0 + x) \simeq B_y(r_0) + x\frac{\partial B_y}{\partial x} \cdot \frac{r_0}{r_0} \cdot \frac{B_y(r_0)}{B_y(r_0)}$$

$$B_y\left(r_0+x\right) \quad \simeq \quad B_y\left(r_0\right)\left(1+\frac{x\frac{\partial B_y}{\partial x}\cdot\frac{r_0}{B_y(r_0)}}{r_0}\right)$$

$$B_y\left(r_0+x\right) \simeq B_y\left(r_0\right)\left(1-\frac{nx}{r_0}\right) \tag{5}$$

Replacing B_y with this function approximation in equation 3 results in

$$e\mathbf{v}B_y\left(r_0\right)\left(1-\frac{nx}{r_0}\right)=m\left(\frac{r_0-x}{r_0^2}\mathbf{v}^2-\ddot{x}\right)$$

$$\left(1-\frac{nx}{r_0}\right)=\frac{m}{e\mathbf{v}B_y\left(r_0\right)}\left(\frac{r_0-x}{r_0^2}\mathbf{v}^2-\ddot{x}\right)$$

From 3we can say $r_0=\frac{mv}{eB_y(r_0)}$ when the orbit is at r_0, so the above becomes

$$1-\frac{nx}{r_0}=\frac{r_0\left(r_0-x\right)}{r_0^2}-\frac{m\ddot{x}}{evB_y\left(r_0\right)}$$

$$\therefore 1=\frac{r_0-x}{r_0}+\frac{nx}{r_0}-\frac{m\ddot{x}}{evB_y\left(r_0\right)}$$

$$1=\frac{r_0-x+nx}{r_0}-\frac{m\ddot{x}}{evB_y\left(r_0\right)}$$

$$1=\frac{r_0+x\left(n-1\right)}{r_0}-\frac{m\ddot{x}}{evB_y\left(r_0\right)}$$

$$1=\frac{r_0}{r_0}+\frac{x\left(n+1\right)}{r_0}-\frac{m\ddot{x}}{evB_y\left(r_0\right)}$$

$$\therefore \frac{m\ddot{x}}{evB_y\left(r_0\right)}=1-1+\frac{x\left(n-1\right)}{r_0}$$

$$\frac{m\ddot{x}}{evB_y\left(r_0\right)r_0}=\frac{x\left(n-1\right)}{r_0^2}$$

$$\frac{mv\ddot{x}}{eB_y\left(r_0\right)r_0}=\frac{v^2x\left(n-1\right)}{r_0^2}$$

Substituting in $r_0 = \frac{mv}{eB_y(r_0)}$ this becomes

$$\frac{r_0}{r_0}\ddot{x} = \frac{v^2 x\,(n-1)}{r_0^2}$$

$$\therefore \ddot{x} = -\left(\frac{v^2\,(1-n)}{r_0^2}\right) x$$

From the chain rule of differentiation $\frac{d^2x}{dz^2} = \frac{d^2x}{dt^2} \cdot \frac{d^2t}{dz^2}$ and, since $\frac{dz}{dt} = v$, $\frac{dt}{dz} = \frac{1}{v}$ and thus $\frac{d^2t}{dz^2} = \frac{1}{v^2}$.

By denoting differentials with regards to z with a prime ($\prime$)

$$x'' = \frac{\ddot{x}}{v^2}$$

$$x'' = -\left(\frac{1-n}{r_0^2}\right) x$$

For a stable orbit $n < 1$ and acceleration is directed towards the central orbit distance. This gives an equation of the form

$$x'' = -\omega^2 x \tag{6}$$

where $\omega = \sqrt{\frac{1-n}{r_0^2}}$ which is simple harmonic motion.

This means that x must have the solutions

$$x = A\cos\left(\frac{\sqrt{1-n}}{r_0} \cdot z\right) + B\sin\left(\frac{\sqrt{1-n}}{r_0} \cdot z\right)$$

Differentiating x with regards to z gives

$$x' = -A\frac{\sqrt{1-n}}{r_0}\sin\left(\frac{\sqrt{1-n}}{r_0}z\right) + B\frac{\sqrt{1-n}}{r_0}\cos\left(\frac{\sqrt{1-n}}{r_0}z\right)$$

Let the initial state at $z = 0$ be $x = x_1$ and $x' = x_1'$, thus

$$x_1 = A\cos 0 + B\sin 0$$

$$\therefore x_1 = A$$

$$x_1' = -A\frac{\sqrt{1-n}}{r_0}\sin 0 + B\frac{\sqrt{1-n}}{r_0}\cos 0$$

$$x_1' = B\frac{\sqrt{1-n}}{r_0}$$

$$\therefore B = \frac{x_1' r_0}{\sqrt{1-n}}$$

Now the above equations become

$$x = x_1 \cos\left(\frac{\sqrt{1-n}}{r_0} \cdot z\right) + \frac{x_1' r_0}{\sqrt{1-n}} \sin\left(\frac{\sqrt{1-n}}{r_0} \cdot z\right)$$

and

$$x' = -x_1 \frac{\sqrt{1-n}}{r_0} \sin\left(\frac{\sqrt{1-n}}{r_0} z\right) + \frac{x_1' r_0}{\sqrt{1-n}} \frac{\sqrt{1-n}}{r_0} \cos\left(\frac{\sqrt{1-n}}{r_0} z\right)$$

$$x' = -x_1 \frac{\sqrt{1-n}}{r_0} \sin\left(\frac{\sqrt{1-n}}{r_0} z\right) + x_1' \cos\left(\frac{\sqrt{1-n}}{r_0} z\right)$$

These now express the approximate position (x) and divergence ($x\prime$) at a position z around the ring, relative to a previous position (x_1) and divergence ($x_1\prime$). Rather than the above formulaic representation, with the initial values encoded into the formula, a matrix representation splits these values away from the constants to give

$$\begin{bmatrix} x_2 \\ x_2' \end{bmatrix} = \begin{bmatrix} \cos\left(\sqrt{1-n}\frac{z}{r_0}\right) & \frac{r_0}{\sqrt{1-n}} \sin\left(\sqrt{1-n}\frac{z}{r_0}\right) \\ -\frac{\sqrt{1-n}}{r_0} \sin\left(\sqrt{1-n}\frac{z}{r_0}\right) & \cos\left(\sqrt{1-n}\frac{z}{r_0}\right) \end{bmatrix} \begin{bmatrix} x_1 \\ x_1' \end{bmatrix}$$

Since approximations have been made in the construction of this result, the smaller the value of z used, the more accurate the result will be. Taking an infinitesimal section of length dz gives us the best approximation

$$\begin{bmatrix} x_2 \\ x_2' \end{bmatrix} = \begin{bmatrix} \cos\left(\sqrt{1-n}\frac{dz}{r_0}\right) & \frac{r_0}{\sqrt{1-n}} \sin\left(\sqrt{1-n}\frac{dz}{r_0}\right) \\ -\frac{\sqrt{1-n}}{r_0} \sin\left(\sqrt{1-n}\frac{dz}{r_0}\right) & \cos\left(\sqrt{1-n}\frac{dz}{r_0}\right) \end{bmatrix} \begin{bmatrix} x_1 \\ x_1' \end{bmatrix}$$

3.2 Vertical Motion

A similar thread can be used to derive a matrix representation of the vertical (y) motion.

Using the same binomial expansion method for the **B** field as in the radial case, the perpendicular (x) component is found

$$B_x(y) \simeq y \frac{\partial B_x}{\partial y}$$

From equation 1 it is known that $\frac{\partial B_x}{\partial x} = -\frac{\partial B_y}{dy}$ and that $\nabla \cdot \mathbf{B} = 0$. Since **B** cannot depend on z, since the ring is symmetric, it can only depend on x and y, thus

$$\frac{\partial B_x}{\partial y} = \frac{\partial B_y}{\partial x}$$

$$\therefore n = -\frac{r_0}{B_y} \cdot \frac{\partial B_y}{\partial x}$$

$$= -\frac{r_0}{B_y} \cdot \frac{\partial B_x}{\partial y}$$

$$\therefore \frac{\partial B_x}{\partial y} = -\frac{nB_y}{r_0}$$

Substituting this into equation 2, perpendicular to the vertical, gives

$$m\ddot{y} = B_x\left(y\right) ev$$

$$= y\frac{\partial B_x}{\partial y} ev$$

$$= -\frac{yevnB_y}{r_0}$$

$$\therefore \ddot{y} = -\frac{evB_y n}{r_0 m}$$

$$= -\frac{ev^2 B_y n}{r_0 mv}$$

Since $r_0 = \frac{mv}{eB_y}$ this becomes
$\ddot{y} = -yn\frac{v^2}{r_0^2}$
$= -\left(\frac{nv^2}{r_0^2}\right)$y
Again, from the chain rule, $\frac{d^2y}{dz^2} = \frac{d^2y}{dt^2} \cdot \frac{d^2t}{dz^2} = \frac{d^2y}{dt^2} \cdot \frac{1}{v^2}$, so

$$y'' = -\left(\frac{n}{r_0^2}\right) y \tag{7}$$

This is once again simple harmonic motion for $n > 0$, with general solution

$$y = C\cos\left(\sqrt{n}\frac{z}{r_0}\right) + D\sin\left(\sqrt{n}\frac{z}{r_0}\right)$$

$$\therefore y' = -C\frac{\sqrt{n}}{r_0}\sin\left(\sqrt{n}\frac{z}{r_0}\right) + D\frac{\sqrt{n}}{r_0}\cos\left(\sqrt{n}\frac{z}{r_0}\right)$$

Where C and D can be calculated in the same way as in the radial case, yielding a transformation matrix

$$\begin{bmatrix} y_2 \\ y_2' \end{bmatrix} = \begin{bmatrix} \cos\left(\sqrt{n}\frac{z}{r_0}\right) & \frac{r_0}{\sqrt{n}}\sin\left(\sqrt{n}\frac{z}{r_0}\right) \\ -\frac{\sqrt{n}}{r_0}\sin\left(\sqrt{n}\frac{z}{r_0}\right) & \cos\left(\sqrt{n}\frac{z}{r_0}\right) \end{bmatrix} \begin{bmatrix} y_1 \\ y_1' \end{bmatrix}$$

Which for an infinitesimal section dz becomes

$$\begin{bmatrix} y_2 \\ y_2' \end{bmatrix} = \begin{bmatrix} \cos\left(\sqrt{n}\frac{dz}{r_0}\right) & \frac{r_0}{\sqrt{n}}\sin\left(\sqrt{n}\frac{dz}{r_0}\right) \\ -\frac{\sqrt{n}}{r_0}\sin\left(\sqrt{n}\frac{dz}{r_0}\right) & \cos\left(\sqrt{n}\frac{dz}{r_0}\right) \end{bmatrix} \begin{bmatrix} y_1 \\ y_1' \end{bmatrix}$$

3.3 Combined Matrix

In some implementations, the radial and vertical matrices have been combined. This is easily achieved by combining the radial and vertical values into a single vector $\begin{bmatrix} x \\ x' \\ y \\ y' \end{bmatrix}$ and extending the transformation matrix to be the following 4×4, matrix

$$\begin{bmatrix} \cos\left(\sqrt{1-n}\frac{dz}{r_0}\right) & \frac{r_0}{\sqrt{1-n}}\sin\left(\sqrt{1-n}\frac{dz}{r_0}\right) & 0 & 0 \\ -\frac{\sqrt{1-n}}{r_0}\sin\left(\sqrt{1-n}\frac{dz}{r_0}\right) & \cos\left(\sqrt{1-n}\frac{dz}{r_0}\right) & 0 & 0 \\ 0 & 0 & \cos\left(\sqrt{n}\frac{dz}{r_0}\right) & \frac{r_0}{\sqrt{n}}\sin\left(\sqrt{n}\frac{dz}{r_0}\right) \\ 0 & 0 & -\frac{\sqrt{n}}{r_0}\sin\left(\sqrt{n}\frac{dz}{r_0}\right) & \cos\left(\sqrt{n}\frac{dz}{r_0}\right) \end{bmatrix}$$

3.4 Motion in Free Space

In free space, ie. in a section without surrounding magnets, the field index n becomes zero. This simplifes the transformation matrix to

$$\begin{bmatrix} x_2 \\ x_2' \\ y_2 \\ y_2' \end{bmatrix} = \begin{bmatrix} \cos\left(\frac{dz}{r_0}\right) & r_0\sin\left(\frac{dz}{r_0}\right) & 0 & 0 \\ -\frac{1}{r_0}\sin\left(\frac{dz}{r_0}\right) & \cos\left(\frac{dz}{r_0}\right) & 0 & 0 \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_1' \\ y_1 \\ y_1' \end{bmatrix}$$

3.5 Phase-Space Shape

From equation 6 it is possible to derive the shape of the rings' acceptance, since we know this simple harmonic motion can have the following solution

$$x = A\sin\left(\frac{\sqrt{1-n}}{r_0^2}z + \phi\right)$$

$$\therefore x' = A\frac{\sqrt{1-n}}{r_0}\cos\left(\frac{\sqrt{1-n}}{r_0^2}z+\phi\right)$$

From these we can derive the following

$$\frac{x}{A} = \sin\left(\frac{\sqrt{1-n}}{r_0}z+\phi\right)$$

$$\frac{r_0}{\sqrt{1-n}}\frac{x'}{A} = \cos\left(\frac{\sqrt{1-n}}{r_0}z+\phi\right)$$

$$\left(\frac{x}{A}\right)^2 = \sin^2\left(\frac{\sqrt{1-n}}{r_0}z+\phi\right)$$

$$\left(\frac{r_0}{\sqrt{1-n}}\frac{x'}{A}\right)^2 = \cos^2\left(\frac{\sqrt{1-n}}{r_0}z+\phi\right)$$

Adding these last two equations together yields

$$\left(\frac{x}{A}\right)^2 + \left(\frac{r_0}{\sqrt{1-n}}\frac{x'}{A}\right)^2 = \sin^2\left(\frac{\sqrt{1-n}}{r_0}z+\phi\right) + \cos^2\left(\frac{\sqrt{1-n}}{r_0}z+\phi\right) = 1$$

$$\frac{x^2}{A^2} + \frac{r_0^2\left(x'\right)^2}{A^2\left(1-n\right)} = 1 \tag{8}$$

Since this is simple harmonic motion, from equation 6 we can say $\frac{1-n}{r_0^2} = \omega^2 = \left(\frac{2\pi}{\lambda}\right)^2$ where λ is the wavelength of the oscillation. Using this we can transform the above equation in to

$$\frac{x^2}{A^2} + \frac{\left(x'\right)^2}{A^2\left(\frac{2\pi}{\lambda}\right)^2} = 1$$

If we call the wavelength $2\Lambda \equiv \lambda$ then

$$\frac{1-n}{r_0^2} = \left(\frac{2\pi}{\lambda}\right)^2$$

$$\frac{1-n}{r_0^2} = \left(\frac{2\pi}{2\Lambda}\right)^2$$

$$\frac{1-n}{r_0^2} = \left(\frac{\pi}{\Lambda}\right)^2$$

$$\frac{1-n}{r_0^2} = \frac{\pi^2}{\Lambda^2}$$

$$\frac{r_0^2}{1-n} = \frac{\Lambda^2}{\pi^2}$$

Substituting this in to equation 8 gives

$$\frac{x^2}{A^2} + \frac{\Lambda^2 \left(x'\right)^2}{A^2\pi^2} = 1$$

This is an equation of the form $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$ which describes an ellipse, in this case in the dimensions x and x', ie. phase space.

The same can be done vertically starting from equation 7

$$y'' = -\frac{n}{r_0^2} y$$

$$\therefore y = C \sin\left(\frac{\sqrt{n}}{r_0} z + \Omega\right)$$

$$\therefore y' = \frac{\sqrt{n}}{r_0} C \cos\left(\frac{\sqrt{n}}{r_0} z + \Omega\right)$$

From these we can say

$$\frac{y}{C} = \sin\left(\frac{\sqrt{n}}{r_0} z + \Omega\right)$$

$$\frac{y' r_0}{\sqrt{n} C} = \cos\left(\frac{\sqrt{n}}{r_0} z + \Omega\right)$$

$$\frac{y^2}{C^2} = \sin^2\left(\frac{\sqrt{n}}{r_0} z + \Omega\right)$$

$$\frac{r_0^2 \left(y'\right)^2}{nC^2} = \cos^2\left(\frac{\sqrt{n}}{r_0} z + \Omega\right)$$

Adding these gives

$$\frac{r_0^2 \left(y'\right)^2}{nC^2} + \frac{y^2}{C^2} = \sin^2\left(\frac{\sqrt{n}}{r_0} z + \Omega\right) + \cos^2\left(\frac{\sqrt{n}}{r_0} z + \Omega\right) = 1 \qquad (9)$$

From equation 7 it is known, from simple harmonic motion, that $\frac{n}{r_0^2} = \left(\frac{2\pi}{\xi}\right)^2$ where ξ is the wavelength of the oscillations. Let $2\Xi \equiv \xi$ then

$$\frac{n}{r_0^2} = \left(\frac{2\pi}{\xi}\right)^2$$

$$\frac{n}{r_0^2} = \left(\frac{2\pi}{2\Xi}\right)^2$$

$$\frac{n}{r_0^2} = \frac{\pi^2}{\Xi^2}$$

$$\frac{r_0^2}{n} = \frac{\Xi^2}{\pi^2}$$

Substituting this into equation 9 gives the following

$$\frac{\Xi^2 \left(y'\right)^2}{\pi^2 C^2} + \frac{y^2}{C^2} = 1$$

This is also an elliptical equation of the form $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$.

Appendix B: Final Source Code

The following is the complete source code, in Python, of the simulation's final incarnation. This is available electronically from its Git distributed version control repository at http://github.com/Warbo/ParticleStorageRingSimulation/tree/master.

Matrix.py

```
#!/usr/bin/env python

class Matrix:
    """This custom class is a pure Python implementation of matrices."""

    def __init__(self, rows, columns):
        self.row_number = rows

        self.column_number = columns

        self.matrix = []
        for r in range(self.row_number):
            self.matrix.append(None)
```

```
        for row in range(self.row_number):
            new_column = []
            for column in range(self.column_number):
                new_column.append(0.0)
            self.matrix[row] = new_column

    def __setitem__(self, position, value):
        pos = position.split(',')
        row = int(pos[0])
        column = int(pos[1])
        self.matrix[row][column] = value

    def __getitem__(self, position):
        pos = position.split(',')
        row = int(pos[0])
        column = int(pos[1])
        return self.matrix[row][column]

    def make_matrix(self, values):
        """Generates a matrix with values given in the (row-major)
        nested list given."""
        rows = len(values)
        columns = len(values[0])
        matrix = Matrix(rows, columns)
        for r, row in enumerate(values):
            for c, element in enumerate(row):
                matrix[str(r)+','+str(c)] = element
        return matrix

    def get_column(self, index):
        column = []
        for row in self.matrix:
            column.append(row[index])
        return column

    def __mul__ (self, other):
        """Returns a Matrix of the result of multiplying this matrix
        with the given matrix."""

        # First check that multiplication is defined for these matrices
        if not self.column_number == other.row_number:
            raise ValueError("Matrix dimensions do not match.")

        # Now make the resulting matrix
        matrix = Matrix(self.row_number, other.column_number)
```

```
            # Now calculate each element of the new matrix
            for r, row in enumerate(matrix.matrix):
                for c, element in enumerate(row):

                    # The current element is the sum of the elements of
                    # this matrix's row and the other matrix's column
                    for i in range(self.row_number):
                        matrix[str(r)+','+str(c)] += \
                            self.matrix[r][i] * other.get_column(c)[i]

            return matrix

        def __str__(self):
            return_string = "[\n"
            for row in self.matrix:
                return_string = return_string + ' ' + str(row) + '\n'
            return_string = return_string + ']'
            return return_string

    m = Matrix(1, 1)
```

Particle.py

```
    #!/usr/bin/env python

    import Matrix

    class Particle(object):

        def __init__(self, matrix):
            self.matrix = matrix
            self.lost = [False, False]

        def make_particle(self, x, y, x_divergence, y_divergence):
            return Particle(Matrix.m.make_matrix([[x],[x_divergence],[y],[y_divergence]]))

        def __str__(self):
            return 'P' + str(self.matrix)

    p = Particle(Matrix.m)
```

RingSection.py

```
#!/usr/bin/env python
import Particle

class RingSection(object):

    def __init__(self, matrix):
        self.matrix = matrix

    def __mul__(self, particle):
        lost = particle.lost
        to_return = Particle.Particle(self.matrix * particle.matrix)
        to_return.lost = lost
        return to_return
```

Ring.py

```
#!/usr/bin/env python
import RingSection
import Matrix
import Particle

class EscapeException(Exception):
    pass

class Ring(object):

    def __init__(self, ring_sections, particles, xmax, ymax):
        self.particles = []
        self.particles.append(particles)
        self.sections = ring_sections
        self.xmax = xmax
        self.ymax = ymax

    def step(self, steps):
        ps=[]
        for particle in self.particles[-1]:
            steps_taken = 0
            try:
                for section in self.sections:
                    if steps_taken >= steps:
                        raise EscapeException()
                    ps.append(section * particle)
                    if self.is_escaped(particle):
```

```
                        raise EscapeException()
                    steps_taken += 1
            except EscapeException:
                pass
            self.particles.append(ps)
            ps=[]

    def is_escaped(self, particle):
        if abs(particle.matrix['0,0']) > self.xmax:
            particle.lost[0] = True
        if abs(particle.matrix['2,0']) > self.ymax:
            particle.lost[1] = True
        return any(particle.lost)
```

Simulation.py

```
import Ring
import RingSection
import Particle
import Matrix
import math
import random

def make_particles(number, xmax, ymax, xpmax, ypmax):
    particles = []
    for index in range(number):
        particles.append(Particle.p.make_particle(\
            (random.random()*2*xmax)-xmax, \
            (random.random()*2*ymax)-ymax, \
            (random.random()*2*xpmax)-xpmax, \
            (random.random()*2*ypmax)-ypmax))
    return particles

def count_x_lost(particles):
    count = 0
    for particle in particles:
        if particle.lost[0]:
            count += 1
    return count

def count_y_lost(particles):
    count = 0
    for particle in particles:
        if particle.lost[1]:
            count += 1
```

```
    return count

def count_lost(particles):
    count = 0
    for particle in particles:
        if any(particle.lost):
            count += 1
    return count

# Set some properties
n_steps = 20
particles = 10
ring_steps = 40.
total_steps = ring_steps * 50
r0 = 2.
xmax = 0.03
ymax = 0.03
xpmax = xmax/r0
ypmax = ymax/r0

dz = (2 * r0 * math.pi) / ring_steps

# These are the field indices to use
ns = [float(x)/n_steps for x in range(1, n_steps)]

# This stores the rings to be simulated
rings = []

# Make one ring per field index
for n in ns:
    # Create the matrices for this field index
    ring_matrix = Matrix.m.make_matrix([\
        [math.cos(((1-n)**0.5)*(dz/r0)), (r0/((1-n)**0.5))*math.sin(((1-n)**0.5)*(dz/r0)),0.
        [-1. * (((1-n)**0.5)/r0)*math.sin(((1-n)**0.5)*(dz/r0)), math.cos(((1-n)**0.5)*(dz/r
        [0.0,0.0,math.cos((n**0.5)*(dz/r0)), (r0/(n**0.5))*math.sin((n**0.5)*(dz/r0))],\
        [0.0,0.0,-1. * ((n**0.5)/r0)*math.sin((n**0.5)*(dz/r0)), math.cos((n**0.5)*(dz/r0))]
        ])

    empty_matrix = Matrix.m.make_matrix([\
        [math.cos(dz/r0), r0*math.sin(dz/r0),0.0,0.0],\
        [(-1.0/r0)*math.sin(dz/r0), math.cos(dz/r0),0.0,0.0],\
        [0.0,0.0,1.0,dz],\
        [0.0,0.0,0.0,1.0]\
        ])

    # Make the sections for this ring
```

```
    sections = []
    for x in range(int(ring_steps)):
        sections.append(RingSection.RingSection(ring_matrix))

    # Make the ring
    rings.append(Ring.Ring(sections, make_particles(particles, xmax,
ymax, xpmax, ypmax), xmax, ymax))

# Run the simulation
for x, ring in enumerate(rings):
    ring.step(total_steps)
    print str(len(rings) - x)

# Receive the results
out = ["n, remaining, xlost, ylost"]

for x, ring in enumerate(rings):
    out.append(str(ns[x]) + ', ' + str(particles - count_lost(ring.particles))
+ ', ' + str(particles - count_x_lost(ring.particles)) + ', ' + str(particles
- count_y_lost(ring.particles)))

# Write the results to a file
outfile = open("OUT.csv", "w")
for line in out:
    outfile.write(line + '\n')
outfile.close()
```

4 References

[1] Banford, A. P., *The Transport of Charged Particle Beams*, Spon, 1966

[2] Hamkins, J. D., *Infinite Time Turing Machines: Supertask Computation*, arXiv:math/0212047v1, 2002

[3] Durrett, R., *Probability: Theory and Examples*, http://www.math.cornell.edu/~durrett/PTE/pte.html, 2009