The Joys of Writing Code in Only One Line

Hugo Robinson

Contents

Analysis 1

Background 1

Smaller Than Indivisible? 1

Crash Course Physics 1

You Must Be Joking Mr Feynman 1

The Standard Model 2

Particles and Anti-particles 2

When Small Things go Bang 2

A Small Side Note on the Creation of the Universe 2

Description of project 3

Inside a Simulation 3

Particles from Top to Bottom 3

Why Functional? 1

You Must be Joking Mr McCarthy 1

A Cup of Coffee? 1

Challenges - Imagine Programing Without, Variables, Loops, Selection or Sequence 1

What is the Point? 2

Approach 2

Specific objectives 2

Problem analysis 3

Design 5

Overall design approach 5

Plan for final development structure 5

The Class System 5

Like an Onion 1

Why C# and not Haskell 1

Specific problems and their solutions 1

Heads and tails 1

How to Randomise a Determined Value 2

Generic Parameters 2

One of these is Not Like the Others 2

Areas involving technical complexity 3

Selection 3

Nested Selection 4

Recursion 4

Selection, Recursion with Functional Lists 4

Map 5

Fold 5

Filter 5

Tuples 5

How to do Functional in c# 5

Technical implementation 6

Testing 7

Test Plan **Error! Bookmark not defined.**

High Level test (Outside in) 7

Low Level Test Plan (Inside out) 9

Evaluation 14

# Analysis

## Background

### Smaller Than Indivisible?

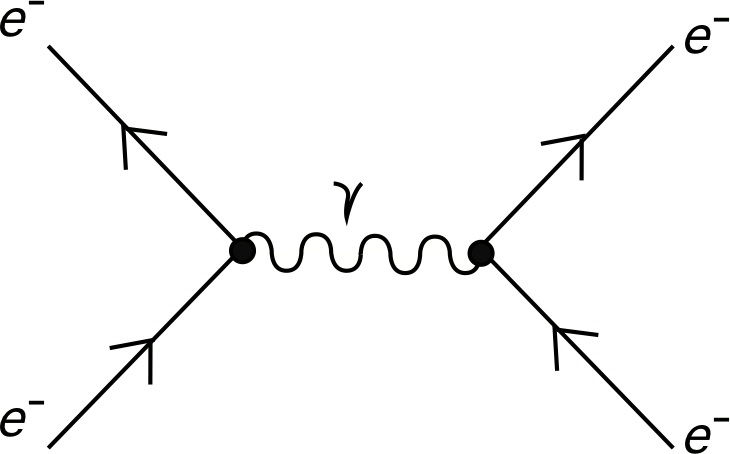
The world of physics can seem big and complex, and quite frankly it is. However everything form the smallest amoeba to the largest Galaxies all work on the same principles, the interactions between particles. Particle physics, as we know it today was first discovered between 1879 and 1897 when the electron was discovered. This meant that the atom was not indivisible but was comprised of many smaller particles. The discovery led Joseph Thompson to release his Plum Pudding representation of an atom where a dense cloud of positive charge was interlaced with electrons. However as we know today, this model is not correct, the new model with a nucleus of positive charge was proposed Ernest Rutherford in 1919 after his experiment which involved firing alpha particles at a very thin sheet of gold foil. Most of the particles passed through the foil as expected however some diffracted and even fewer bounced back. This was so astonishing at the time that Ernest Rutherford said, “It was quite the most incredible event that has ever happened to me in my life. It was almost as incredible as if you fired a 15-inch shell at a piece of tissue paper and it came back and hit you” <https://www.goodreads.com/quotes/105980-it-was-quite-the-most-incredible-event-that-has-ever>. Then in 1931 James Chadwick discovers the neutron, which gives the final piece of the puzzle for the creation of the atom and how the standard model of the atom is represented today. As a result of these studies the four fundamental forces of nature were discovered, The Strong Force, The Weak Force, Electromagnetism and Gravity. The Strong force mediates how the nucleons stay together and acts over a range of femptometers. The weak force mediates the decay of particles. The electromagnetic force mediates how different charges and magnetisms interact with each other, and is the reason matter can clump together to form galaxies, stars and planets.

### Crash Course Physics

Stuff From A level textbook

### You Must Be Joking Mr Feynman

Along with this, throughout the last century, different interactions between particles have been observed, both natural and created in colliders. These interactions govern everything from the levitation of superconductors <http://www.superconductors.org/> to the entire field of Chemistry. Many of these interactions can be represented through Feynman diagrams.



This Feynman Diagram shows a basic electrostatic repulsion where two electrons that poses the same charge come too close together. Then an interaction particle, in this case a photon, causes the two electrons to move away from each other. This diagram is also represented on a graph where the x-axis is space and the y-axis is time.

### The Standard Model

### Particles and Anti-particles

Everything we see around us is matter, from the tables I am working on, to the chair I am sitting on, to the snow that is drifting gently outside. All of it is created from particles; however not all matter in the universe is created from the same particles, anti-matter also exists and it is made from anti-particles. These anti-particles are in every way that same as their counterparts however they have an opposite charge. So the standard particle an electron has a charge of -1, its anti-particle, the positron has a charge of +1 but they both have the same rest mass (9.10956 x 10 -31 Kg).

### When Small Things go Bang

One of the simplest interactions that is found in nature is the annihilation. This is when a particle and its antiparticle collide to create two photons of equal energy and opposite directions. Dependent on the velocity of the two particles the energy of the photons will be different, but the two photons will always have the same energy as each other. This is calculated by using Einstein famous equation E=mc2 (E = Energy, m = Rest Mass, c = The speed of light) and the equation Ke = 0.5mv2 (Ke = Kinetic Energy, m = Rest Mass, v = Velocity). The addition of E and Ke gives the total energy for that particle. This energy is then halved and as photos have a set speed (3x108) the energy difference takes the form of a higher frequency for more energy and a lower frequency for less. This allows photons to be created in any part of the electro-magnetic spectrum, from radio to gamma.

### A Small Side Note on the Creation of the Universe

2 Paragraphs

Nothing to do with the project, just motion the background about particle physics

## Description of project

### Inside a Simulation

This project is designed to simulate particle interactions in multiple scenarios where up to two particles are entered by the user, along with some requirements such as velocity. The program will then use this information to run real calculations to determine the velocity and position of new particles created. This information will then be presented to the user.

### Particles from Top to Bottom

Particle classification is a great example of a hericical structure where each individual particle is represented by a group classification.

Not FP in this section

Particle Hierarchy

1 paragraph

To create a particle interaction simulator

## Why Functional?

[If not already is your opportunity to persuade the examiner that this really is an A-level standard project. You might not need this section]

### You Must be Joking Mr McCarthy

In 1958, John McCarthy was leading a team at MIT to ‘create a system for programming computations over symbolic data’ <https://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf> - Chapter 2 Lisp - Page 5.

### A Cup of Coffee?

#### The analogy used here was used by Luca Bolognese in a talk about Microsoft’s F# language in 2009. He compares the execution of statements and expressions to making a cup of coffee. In a procedural language the program is written as a sequence of commands, each command dictates what to do next to reach the end result. This can involve calculations or the modification of objects. To bring it back to the coffee analogy, to obtain a coffee with two sugars in the program would first create the coffee object and then add two sugars to it to reach the final result. However Functional languages are written using the evaluation of expressions, these expressions dictate the properties of the object instead of creating it first with standard properties, this means the program will not create a coffee and then add the sugar but will instead create a coffee with sugar already in. This means we cannot use the incorrect form of the object as there is only one form, or alternatively wee cannot drink the coffee without sugar as it is already created with sugar in. This subtle difference in how the coffee is created has some large effects on the approach on how to write the code to do so. Going functional can pose some serious challenges, but also bestows some excellent rewards. Book- Page 5

##### 

Some history on functional

Lambda

Lisp

Use book?

https://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf

### Challenges - Imagine Programing Without, Variables, Loops, Selection or Sequence

There are some very serious challenges that arise when trying to create a program using functional techniques. One is the lack of variables used. Right from the beginning of being taught to program variables have been used, however they are not allowed in functional as variables can change. Whereas in functional all types have to be immutable. Therefore if any data structure needs to be changed, a new data structure has to be created with exactly the same entities as the previous one and whatever change has taken place also applied. The lack of sequence also makes functional difficult as everything has to be done using only one line within a function, this reduces some of the clarity when coding and requires simple sequential statements to either be moved into one line, or extracted into separate functions. There is also no selection in functional, therefore ternary operators must be used on the return statement instead. Any fort of loop is also not allowed in functional programing; therefore recursion must be used throughout.

Referential Transparency

### What is the Point?

The Functional Paradigm is not new, but until recently it has not be used outside academia however, with todays requirements of big data it has become much more prevalent. It is used heavily in science as functional techniques due to its reliability when headlining large volumes of data; it also can be very easily tested to make sure the results that are output are correct and there are no logic errors. Having learnt some basics of functional programing through the course I decided I wanted to peruse it further, extending myself past the trivial problems introduced to us. Therefore with my interest in the topic and its evident use in science it was perfect for me to develop me project around.

Know benefits

Terseness (Lots of functionality in low code)

High complexity in small amount of code

“I didn’t have time to write you a short letter, so I wrote a long one instead” – mark Twain

All of the problems with the functional paradigm

Need to reference Books in this section

## Approach

External stakeholders -Business approach

My approach is to present the idea of what I am trying to do to a physics teacher who will become the stakeholder in my project. I will then meet with them to gain ideas for added interactions to my project. This approach will allow me to develop my program in line with the requirements of the stakeholder. The requirements of the stakeholder will be reasonably restricted by the functional nature of the program.

Show that ES understood what the program does

Gave specific examples for functions wanted

Give evidence for talks (Notes)

[*If* you have a specific stakeholder, include notes from discussions here]

## Specific objectives

Your solution should meet *most* of your objectives, but it is OK to list *some* objectives that are not met

MoSCoW Analysis

Must have

My main objective for this project was to implement the core if the program using functional programing techniques.

The program should allow a user to input what interactions they would like to simulate and the program will the return the list of particles created and some necessary properties of those particles (eg velocity)

A range of particles implemented as objects with their own set porpities that the rest of the program can call upon.

The stakeholder has provided some requirements for simulations he would like to see and these need to be met

* Add Decays W+ and W-
* Ve interactions
* Electrostatic repulsion
* Pair Production
* Cylcratron/Velocity Selector
* Vectors for particles ejected

Should have

Full list of elements that can be called upon when an element is created

Full headachy of different sub atomic particles.

Could have

To create long decay chains using the decay graph for heavy, unstable elements such as uranium until they become stable.

Decides if an element created is stable or unstable and then gives the option for the element to undergo its decay process.

Won’t have

Bubble chamber GUI due to functional

Large Hadron Collider GUI due to functional + processing

Functional programing (Reference functional paradigms)

Information on how particle interactions work

Meet requirements of a stakeholder

## Problem analysis

This is a description of *what* needs to be implemented, not *how*]

As the program is physics based there is quite a lot of maths involved, which if implemented incorrectly could cause large logic errors.

Particle interactions are usually very set in how they will turn out and therefore some parts of the program will seem like the information will have been looked up instead of calculated.

To simulate modern, high energy collisions such as the ones taking place in the Large Hadron Collider where particles like the Higgs Boson are created are far to complicated for both myself and my computer to do, as they require the best equipment at CREN to attempt to simulate a collision like that.

The position of each particle after collision will have to be calculated alongside wits velocity, this will be done using a particles rest mass and velocity

Physics based

How particle interactions work

# Design

## Overall design approach

The original plan – GUI + Functional mideset

### Plan for final development structure

Layer 1: Particle

* Where the particle diagram is used and that relation is coded into the system. All particles are also provided with a full array propitiates and methods they may require (eg Mass to Energy)
* This is the first layer to be implemented as it will allow the particles to be used throughout the development of the rest of the program with ease
* Uses separate program to the collision, vectors and UI programs
* Start with the Quarks and then move onto the bigger particles

Layer 2: Collision

* Where all the calculations for the collisions takes place and outputs what particles are created
* Will need different collisions for different types (eg Electron capture, annihilation, Proton-proton)
* Should not require any of the vectors layer in coding

Layer 3: Vector

* Where the vector calculations and positioning takes place
* Calls upon both the particles and collisions layer to determine weather any new collisions take place after the primary collision

Layer 4: UI

* The terminal screen UI should be developed alongside the rest of the program but should not be integrated into it to allow a possible development of a graphical UI at a later date, however this step will not be included in this project.
* This section will not be following the functional paradigm as it is very challenging to develop a UI as “A functional program defines a pure function, with no side effects, however the whole point of running a program is to have some side effect” – https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/ - Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell – Slide 8

This plan keeps allows for a higher layer to call on a lower layer but not the other way, this is to make sure that further down the development of the program, if a layer needs to be radically changed it should be easier as it is not being referenced by every other part of the program, only the layers above it.

### The Class System

The classification of particles allows for me to easily implement a class hierarchy system, which will form the base for every interaction in the project, as these classes will represent all of the particles with their attributes being stored as properties. These classes can then be created at individual instances multiple times and still retain their basic properties such as charge and rest mass. However their position and velocity can still be changed. A basic version of the particle hierarchy can be seen below.

Where a represents an abstract class and a represents a concrete class.

Particle

Lepton

Quark

Boson

Electron

Up

Photon

Meson

Hadron

Proton

Pion

### Like an Onion

Alongside keeping the layers separate for management it also integrates very well into the functional mind-set. With functions in each layer calling functions a lay this is the talking abouter below it, meaning that a pure function in one layer will always stay pure as it cannot call a dirty function, which will only reside in the highest layer (UI). A pseudo-code example of how this function call would look would be;

*UI(Vector(Collision(Particle(Arguments))))*

Where the arguments are only the values required in the Particle’s constructor, these values would be obtained through the UI layer form user inputs.

*Variable Arguments => User Input*

*Variable CollisionOutputs = > UI(Vector(Collision(Particle(Arguments))))*

*Terminal Output => CollisionOutputs*

Note that Arguments is stored in a variable, this is not allowed in functional and this phrase involves sequence which is also not allowed in functional, however both are necessary for a UI.

Other layers may also require other arguments such as a random number.

*UI(Vector(Collision(Particle(Arguments),RandomNum)))*

### Why C# and not Haskell

Need to redo

As C# is able to support the functional style I thought it best to continue coding in it as it is the language I have used throughout my A-Level. As I have used C# for a while I know how to interact with the UI effectively which is very difficult in Haskell and as my program will not have a clean UI I do not need to spend lots of time on it. Equally C# supports automated testing, this is very compatible with the functional style allowing me to debug easier. On the topic of debugging the Visual Studios IDE allows me to debug the code much easier, especially as I am still unfamiliar with functional. However C# does not check for the functional rules unlike Haskell, therefore I will have to be very diligent that I am sticking to the rules thought the core of the program.

## Specific problems and their solutions

Started but not complete

### Heads and tails

Normal list in C# is simply

Explain normal list

Explain F list

However a list in functional programing is quite different, for a start a list that cannot be changed is a lot less useful than one that can. Therefore how is it implemented. A functional list consists of a head and a tail, where the head is the first item in the list and the tail is the rest of the list.

Using functional library from NuGet

### How to Randomise a Determined Value

FRandom

### Generic Parameters

One major issue that can occur is how to passes functions different permeates at run time and not hardcode them in beforehand. The answer is generics, this allows the program to infer the type of the perimeter at run time meaning that multiple functions that have the same implementation but different perimeter types can be reduced into a single function with generics.

Add example?

Generics

### One of these is Not Like the Others

The problem of the photon has troubled physicist for centuries so it will come as no surprise that it will also cause problems for this project. Due to the photon having no mass, unlike everything else, a constant velocity, unlike everything else and is both a wave and a particle at the same time will make it difficult to incorporate their calculations into functions that’s implementations are designed for all other particles.

There are two solution to this problem, changing all other particles to work using their De Broglie wavelength to match the wave like nature of light or to use overloaded functions. Overloaded functions are functions with the same implementation; however take different perimeters. This means upon runtime if a function that is overloaded is passed a photon it will use the function designed for it and return the appropriate answer. A Pesudo-Code example of an overloaded function to return the combined energy of two of the same particles is as shown.

*CombinedEnergy (Generic Particle1, Generic Particle2)*

*{return(Particle1.RestMass\*c2) + (Particle2.RestMass\*c2)}*

*CombinedEnergy (Photon Particle1, Photon Particle2)*

*{return((h\*c)/Particle1.Wavelength) + (h\*c)/Particle2.Wavelength)}*

The first function uses the equation E = mc2 (E = Energy, m = Rest Mass, c = Speed of Light in a vacuum) to find the energy of a particle however this does not work for photons due to them having no mass and therefore by this equation zero energy.

The second function uses the equation E = h\*c/Wavelength (E = Energy, h = Plank’s Constant, c = Speed of Light in a vacuum) to find the energy of the photon, which will not work for the other particles.

## Areas involving technical complexity

Specific functional problems solved and specific functions used

Change Pseudo -code to screenshots of true code

### Selection

Selection in procedural programing is fairly simple in high level languages with the IF statement being found in most languages. However in functional programing does not allow selection in the form of an If statement:

*IF (Boolean Condition) Then*

*Return a*

*Else*

*Return b*

As this is not allowed and selection is a vital part of my program, I must turn to ternary operators. A ternary operator is a selection statement that executes on one line, therefore it does not break any of the rules of functional program, a ternary operator would be implemented like this:

*Return Boolean Condition ? a : b*

Where everything left of the question mark is the selection itself and the colon acts as the ELSE.

Selection in True Functional

The functional language Haskell also has selection, in the form of guards. Guards are represented in Haskell using the vertical line symbol, | , or “pipes” and follow a similar format to the switch case selection statement:

*Switch (Value)*

*Case(a):*

*Execute code*

*Break;*

*Case (b):*

*Execute code;*

*Break;*

The same thing can be stated in Haskell:

*Function Definition*

*Func (f)*

*| f == a = Execute code*

*| f == b = Execute code*

Notice the pipes act in a very similar way to the case statements shown above however they do not execute in sequence unlike the case statements.

### Nested Selection

With the same line of thought from the section above, nested selection also uses ternary operators in functional, therefore instead of being implemented like they would in sequence:

*IF (Boolean Condition 1) Then*

*IF (Boolean Condition 2) Then*

*Return a*

*Else*

*Return b*

*Else*

*IF (Boolean Condition 3) Then*

*Return c*

*Else*

*Return d*

As a ternary operator it is implemented like so:

*Return Boolean Condition 1 ? Boolean Condition 2 ? a : b : Boolean Condition 3 ? c : d*

*Return Boolean Condition 1 ? Boolean Condition 2 ? a : b : Boolean Condition 3 ? c : d*

Where each colour represents one condition: This functionality allows for multiple selection statement to execute on the same line

### Recursion

One of the major difficulties with functional programing is the inability to use loops, this means no iteration over data, no checking or creation of lists and everything can only be used once before the program has to be restarted, however this can be overcome by using recursion.

Recursion is the process of calling a function from within itself to create iteration in the program. A recursive function will store the result of each pass on the call stack before calling itself with a slightly smaller version of the same problem. Once an end condition is reached where the function will no longer call itself, the call stack is then popped, returning all of the previous results.

### Selection, Recursion with Functional Lists

Ternary operators and recursion work seamlessly with each other. As the selection provides the capability to end the recursion when necessary.

### Map

### Fold

### Filter

### Tuples

### How to do Functional in c#

# Technical implementation

[This is your complete code, with some annotations added to highlight complexity and/or reference sections in Design]

Paste in in colour

Use a class diagram

Need to add subheadings for all classes used

Add GitHub link + screenshot of checkins

# Testing

[Evidence that the system works. Can be a reference to one or more video demos on YouTube. ]

[Evidence that it works robustly, and for multiple scenarios]

[Include a detailed test plan, which should specifically highlight edge/boundary cases and error trapping]

[Emphasis is on broad testing, of the *main* functionality, not peripheral functionality such as ‘log on’]

## Intro paragraph

### This testing will consist of two elements to the testing. The high level testing is testing the user interaction with the system and if it guards against the user inputting values that don’t work. This will involve using the program and entering accepted values and incorrect values, providing screenshots of what is produced. The second form of the testing is the low level testing where unit-tests are used to test each function in the program. This works very well with functional programing as each function can be called and tests individually to allow me to get full coverage throughout the program.

## High Level Test Plan (Outside in)

This section is based on what the user can do and how to safeguard against errors

|  |  |  |  |
| --- | --- | --- | --- |
| What Testing? | What Case? | Expected | Actual? Screenshots? |
| 1. Annihilation | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Proton – Anti proton | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Electron - Positron | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Enter the modulus of the velocity | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| 2. Electron Capture | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Enter mass number | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Enter proton number | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| 3. Pair Production | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Enter Wavelength | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Enter Frequency | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Cyclatron? |  |  |  |
| Atom interactions | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Enter atomic number | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Enter Mass number | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Beta-Plus | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Beta-Minus | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Exit to main menu | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Electrostatic Repulsion WIP | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |
| Exit | Happy |  |  |
|  | Edge |  |  |
|  | Fail |  |  |

UI/Error Handling

## Low Level Test Plan (Inside out)

Using unit tests to do all of these

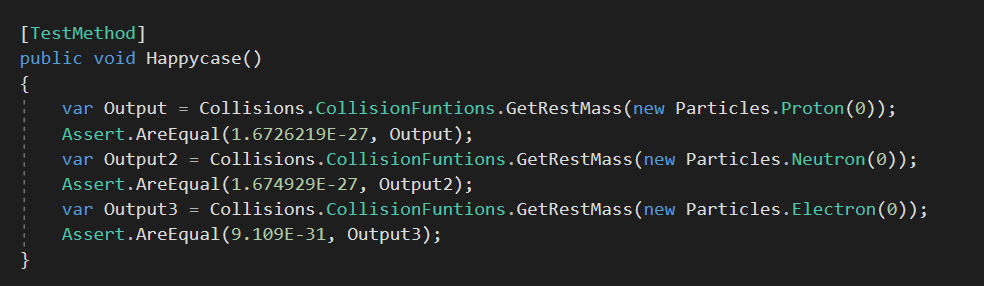
Using an even but structured and extensive testing approach where every function is tested in the same way

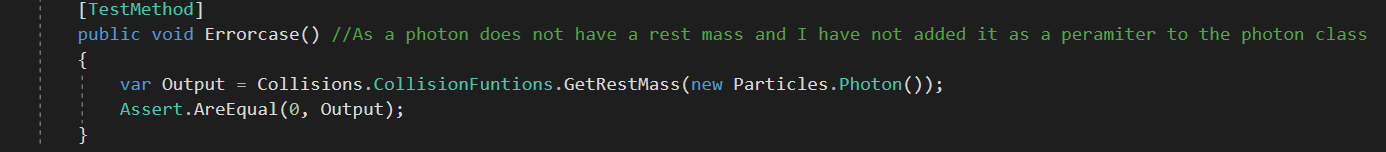
### ***Collision Functions***

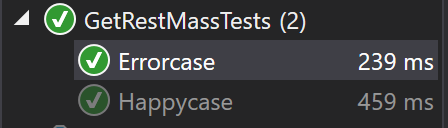
Lowest Functions

static Func<Particles.Particle, double> GetRestMass = x => x.RestMass;

* Normal –



* Error – Pass in an object without a rest mass
* Tests Passing



static Func<Particles.Particle, double> GetVelocity = x => x.Velocity;

* Normal - Pass in a particle which has a velocity
* Edge – N/A
* Error - Pass in an object without a velocity

static Func<double, double, double> add = (x, y) => x + y;

* Normal -
* Edge -
* Error -

public static double TotalRestmass(FList<Double> ListOfMasses)

* Normal – Enter list of Masses
* Edge -
* Error – Enter list of objects

public static double TotalVelocity(FList<double> ListOfVelocity)

* Normal - Enter list of Velocitys
* Edge -
* Error - Enter list of objects

public static double WavelengthToEnergy (Photon Photon)

* Normal – Enter normal wavelength Photon
* Edge – Enter one very low wavelength and one very high
* Error –

public static double FrequencyToEnergy (Photon Photon)

* Normal - Enter normal frequency Photon
* Edge - Enter one very low wavelength and one very high
* Error -

public static double EnergyToVelocity(double energy, double mass)

* Normal -
* Edge -
* Error -

private static double VelocityToEnergy(double totalParticleVelocity, double totalRestMass)

* Normal -
* Edge -
* Error -

private static double MassToEnergy(double totalRestMass)

* Normal -
* Edge -
* Error -

public static double EnergyToWavelength(double energy)

* Normal -
* Edge -
* Error -

public static double EnergyToFrequency(double energy)

* Normal -
* Edge -
* Error –

public static double Calculater(double restMass, int velocity, double fluxDensity, double charge)

* Normal -
* Edge -
* Error –

public static FRandom RandomiseVelocity(FRandom Rand)

* Normal -
* Edge -
* Error –

public static double CalculateVFromR<T>(double Edge, T particle, double FluxDensity) where T : Particle

* Normal -
* Edge -
* Error –

public static string PartialFeynmanDiagram(FList<string> List)

* Normal -
* Edge -
* Error –

public static FList<Proton> CreateProtonList(int AtomicNumber, FList<Proton> AtomicNumberList)

* Normal -
* Edge -
* Error –

public static FList<Neutron> CreateNeutronList(int NeutronNumber, FList<Neutron> NeutronNumberList)

* Normal -
* Edge -
* Error –

private static T CopyWithNewCharge<T>(T p) where T : Particle

* Normal -
* Edge -
* Error –

public static Particle GenerateParticleWithRandomVelocity<T>(T particle, FRandom rand) where T : Particle

* Normal -
* Edge -
* Error –

public static Tuple<Neutron, Positron, ElectronNeutrino> BetaPlusDecayIndividual(Proton P)

* Normal -
* Edge -
* Error –

public static Tuple<Proton, Electron, AntiElectronNeutrino>

BetaMinusDecayIndividual(Neutron N)

* Normal -
* Edge -
* Error –

Middle Functions

public static Photon CreateAnnialationPhoton(Particle Particle, Particle AntiParticle)

* Normal –
* Edge –
* Error -

public static string FeynmanDiagram(FList<String> Inputs, FList<String> Outputs)

* Normal –
* Edge –
* Error -

public static Proton CreatePairproductionOutputGreaterThanProtonRestMass(Photon Photon)

* Normal –
* Edge –
* Error -

public static Electron CreatePairproductionOutputLessThanProtonRestMass(Photon Photon)

* Normal –
* Edge –
* Error -

private static FList<T> GenerateListOfParticlesWithSetVelocities<T>(T particle, FList<double> f) where T : Particle

* Normal –
* Edge –
* Error -

private static bool CheckIfParticleVelocityIsInboundries<T>(T Particle, Tuple<double, double> tuple, double FluxDensity) where T : Particle

* Normal –
* Edge –
* Error -

public static FList<T> ChangeNegativeChargeToPositive<T>(FList<T> fList) where T:Particle

* Normal –
* Edge –
* Error -

public static Tuple<double,double> FindRangeForVelocitiesFromGapWidth<T>(double lowerEdge, double higherEdge,T Particle, double FluxDensity ) where T: Particle

* Normal –
* Edge –
* Error -

public static FList<Particles.Particle> GenerateListOfParticlesWithRandomVelocity<T>(T particle, int numberOfInputParticles, FRandom Rand) where T : Particle

* Normal –
* Edge –
* Error -

public static Atom BetaPlusDecayAtom(Atom A)

* Normal –
* Edge –
* Error -

public static Atom BetaMinusDeacyAtom(Atom A)

* Normal –
* Edge –
* Error -

public static Atom AlphaDecay (Atom A)

* Normal –
* Edge –
* Error –

Heights Functions

public static Tuple<Photon, Photon> Annialation(Particles.Particle Particle, Particles.Particle AntiParticle, FRandom Rand)

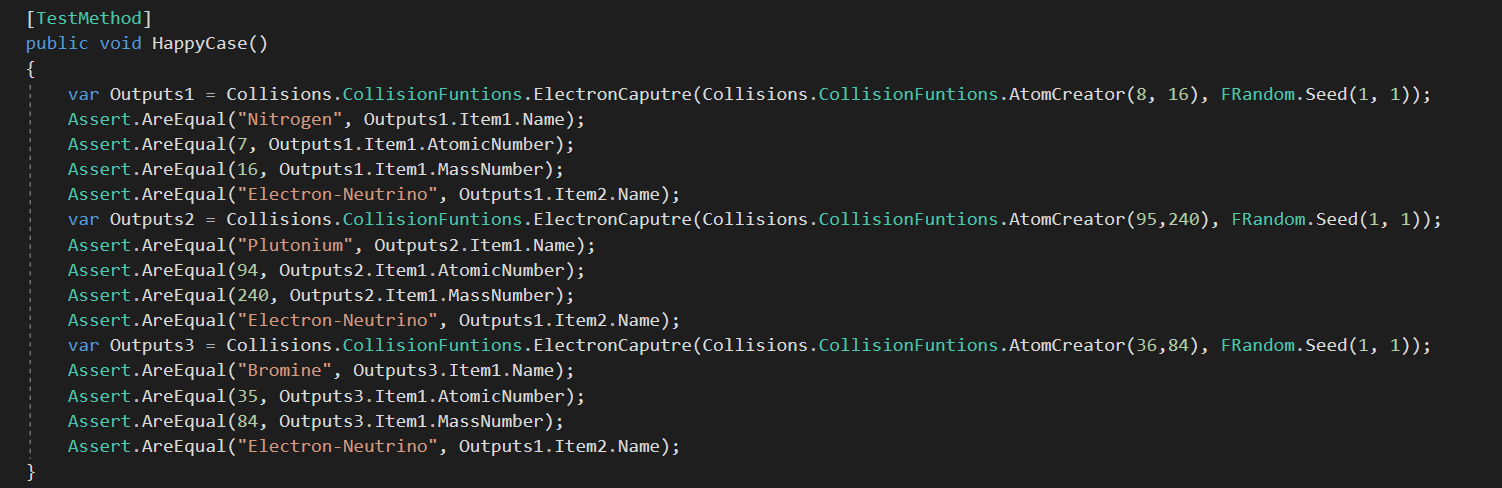
* Normal –
* Edge –
* Error -

public static Tuple<Particle, Particle> PairProductionPhoton(Photon Photon, FRandom Rand)

* Normal –
* Edge –
* Error -

public static Tuple<Atom, Particle> ElectronCaputre(Atom Atom, FRandom Rand)

* Normal –



* Edge –
* Error -

public static FList<Particles.Particle> Cyclatron<T>(T Particle, double FluxDensity , int NumberOfInputParticles, FRandom Rand, double LowerEdge, double HigherEdge) where T : Particle

* Normal –
* Edge –
* Error -

public static Tuple<Particle,Particle> ElectrostaticRepulsion <T>(T P1, T P2, FRandom Rand, Vector3D StartingPositionParticle1, Vector3D StartingPositionParticle2) where T : Particle

* Normal –
* Edge –
* Error -

public static Atom AtomCreator(int AtomicNumber, int MassNumber)

* Normal –
* Edge –
* Error -

### ***Vector Functions***

Lowest Functions

public static Tuple<Vector3D,Vector3D> CreateOpposite (Vector3D V)

* Normal –
* Edge –
* Error -

Middle Functions

public static Particle DistanceEjected<T> (T Particle) where T : Particle

* Normal –
* Edge –
* Error –

public static Photon DistanceEjected(Photon Particle)

* Normal –
* Edge –
* Error –

public static Particle EdgeOFContainmentChecker<T> (T Particle) where T : Particle

* Normal –
* Edge –
* Error -

public static Photon EdgeOFContainmentChecker (Photon Particle)

* Normal –
* Edge –
* Error -

Heights Functions

public static Particle SingularEjection<T>(T p, FRandom Rand) where T : Particle

* Normal –
* Edge –
* Error -

public static Photon SingularEjection(Photon p, FRandom Rand)

* Normal –
* Edge –
* Error –

public static Tuple<Particle,Particle> OppositeEjections<T> (T P1, T P2, FRandom Rand) where T : Particle

* Normal –
* Edge –
* Error –

public static Tuple<Photon,Photon> OppositeEjections(Photon P1, Photon P2, FRandom Rand)

* Normal –
* Edge –
* Error –

public static Particle AdditionOfVectorsForRepulsion <T>(Vector3D StartingPosition,T P1, T P2) where T : Particle

* Normal –
* Edge –
* Error –

Need to write a test plan

Execute the test plan

* High level scenario tests (Does it work for the customer)
* If for certain inputs I will definitely get other outputs
* For randomising need to run allot and show results are to be expected

Do the menus work

Error trapping

Unit tests (Very good with functional) ‘Inside Out’ testing after ‘Outside in’ Testing’

Even testing, not too deep on one section

# Evaluation

How well did you meet your own objectives?

Evidence of *real* feedback from stakeholders and/or real users.?

Evidence of objective ‘reflection’ on your own project.

Further directions.

References