1 Implementation Details

For this project I implemented a single step time-independent generalised Runge-Kutta integrator along with native support for three Runge-Kutta methods, the classic fourth order method, the Dormand-Prince fifth order method (without the final step reuse optimisation), and the Cooper-Verner eighth order method (implemented in `Integrators.hs`). As the n-body kinematics equations are second order (in ODE terms, not numerically) and the Runge-Kutta method is a first order method the implementation itself actually is designed to operate upon the coordinate position and velocity at the same time. To this end I implemented a two dimensional "vector" (mathematical version) data type to represent a two dimensional column vector and additionally a three dimensional version for holding the coordinates themselves. Both types implement a more generalised vector typeclass I had created to encapsulate the mathematical properties required, being a dot product and a way to do scalar multiplication. All other properties of standard normed vector spaces may be derived from there and the typeclass does so. As am implementation detail because both vector types also implement `Num` they have a Hadamard product, which is not usually a property of vectors as used in physics (outside some Quantum Mechanics/Computing things). Scalar multiplication is actually implemented through that as $\alpha\vec{\mathbf{v}} \equiv \alpha\vec{\mathbf{1}} \circ \vec{\mathbf{v}}$. The integrator itself implements their mathematical form more or less exactly as per usual aside from being applied to two 3-vectors at the same time. That is, the integrator implements the following relation for given coefficients a_{ij}, b_i, $1 \leq j \leq i \leq s$:

$$\begin{bmatrix}\mathbf{r}_{n+1}\\ \dot{\mathbf{r}}_{n+1}\end{bmatrix} = \begin{bmatrix}\mathbf{r}_{n}\\ \dot{\mathbf{r}}_{n}\end{bmatrix} + \Delta t \sum_{i=1}^{s} b_i k_i$$

Where for a time independent first order ODE of form $\dot{x} = f(x)$ the k_i values may be calculated as $k_i = f(x_n + \Delta t \sum_j^i a_{ij} k_j)$. For this case since $\dot{\mathbf{r}}$ is already known the f function used in the actual code does not calculate anything and just returns the passed in velocity term there (as found from each acceleration step preceding it). As I chose to construct the k values in a list the main implementation caveat here would be that the a, b coefficients are actually stored in the code "backwards" as a list of lists (within a specific Butcher-Tableu data type) where the first element of the ith list corresponds to k_i instead of k_1.

From a physics perspective solving the second order equations in this way necessarily loses certain geometry considerations and to that end I had originally been working on a multi-step method called the "Gauss-Jackson" method, which is a summed adams family second order integrator. The method itself is mathematically fairly interesting and I had actually spent a decent amount of time on its details but sadly could not finish it in time. Being a second order method it manages to keep some of the geometry considerations, consequently it is fairly standard for calculating satellite orbits at high precision, though it actually requires a single step integrator to start it, which is why the eighth order Cooper-Verner method is implemented.

The rest of the code deals with various physics requirements, such as implementing the acceleration function or representing the orbital parameters of the bodies being simulated. Nothing too complicated, the force equation is in the weak field post-Newtonian form, which adds a small general relativistic correction as outlined in the proposal. The data itself is actually included into the program directly via a Haskell module file, this has to do mainly with the external infrastructure around getting precise orbital data from JPL being fairly involved (all points are stored as

Chebyshev polynomial coefficients) and thus easier to just pull out with existing (Python) code and format for Haskell's use. The data is implemented as a straight list of `OrbitalBody` type, I had experimented with making it a `Data.Array` but the lists actually ended up being marginally faster. I suspect the whole thing is optimised being that it is a constant compile time structure.

The program code (`Main.hs`) then implements the main n-body logic itself, iterate over all points in space and generate their next positions, and repeat. This is also where the parallelisation is implemented. The main way I sought to parallelise this problem, for the n^2 pair-pair form being used here, was to compute each successor point at a time step in parallel. That is, at some time t all points' next position and velocity are already specified so all calculations can be completed in parallel, theoretically at least. In practice I was only able to get Haskell to use two threads for the computation, after getting `rdeepseq` to work on my custom data types. Consequently there is only about a two times speed up from running `nbody` in parallel. In practice I found using four cores offered the fastest times, though I suspect that has to do with my machine being eight cores *hyperthreaded* but this problem being mainly data iteration driven so probably the physical core caches would conflict. I had tried to get Haskell to use more cores, and did succeed actually using a very hacky `parTuple` based solution, but it also ended up making the system spark way too many times so ended up being over all slower. I suspect I am just deeply misunderstanding something with the overall parallel system there in terms of how `Control.Parallel.Strategies` work. Overall the program itself is fairly accurate to real data, and does successfully maintain Mercury's position to around three significant figures across at least two years. It starts becoming too inaccurate at a one day time step around there though and by five years of iteration Mercury has plummeted into the Sun. I have included selected planetary and asteroid data in the submission files as `AllBodiesEpochsData.tex`. While it is formatted as a TEX table the data itself does not fit very legibly so I have not included it in this report itself. General program run time information may be found in `timings.txt` but it is not very well formatted.

(See below for the full code listings)

2 Code Listings

```haskell
{-|
Module      : Physics
Description : Implementation of required physics for N-body problem

Implements a post-newtonian weak-field approximation gravitational acceleration
function and associated vector types required for three dimensional physics.

For more information on Post-Newtonian formulations used herein see [1],
[2] (Eq. 1) and references therein.

[1] /The IAU 2000 Resolutions for Astrometry, Celestial Mechanics, and
↪  Metrology in the Relativistic Framework: Explanatory Supplement/, /Soffel
↪  et al./, 2003. Astronomical Journal, 126:2687-2706.

[2] /Cometary Orbit Determination and Nongravitational Forces/, /Yeomans et
↪  al./

-}

module Physics
       (
         Vec(..)
       , Vec3(..)
       , Vec2(..)
       , pNGRPair
       , pNGRFull
       , CutParams(..)
       ) where

import PhysicsVectors
import PhysicsConstants
import AstroData

-- | Cutoff parameters for GR corrections, 'CutParams' @masscut distcut@
data CutParams a = CutParams a a

chkCutoff :: (Ord f, Floating f) => CutParams f -> OrbitBody f -> OrbitBody f
↪  -> Bool
chkCutoff (CutParams mc dc) source targ
  | mc <= bGM source && stDist <= dc = True
  | otherwise = False
  where
```

```haskell
      stDist = dist (obPos source) (obPos targ)

-- | Computes the pair-wise gravitational acceleration between two bodies
pNGRPair :: (Ord f, Floating f) => OrbitBody f -> OrbitBody f -> CutParams f ->
↪  Vec3 f
pNGRPair source target grcut
  | bid source == bid target = fromScalar 0
  | not $ chkCutoff grcut source target = newton
  | otherwise =
      newton + ((dot vst vst/cAUsq) `scalarMult` newton) + (ms/cAUsq)
      ↪  `scalarMult` grcor'
  where
    rst = (obPos target) - (obPos source)
    ms = bGM source
    (r2, r3) = let r2' = normSq rst in (r2', r2' * sqrt r2')
    newton = (-ms/r3) `scalarMult` rst
    cAUsq = fromRational (cAU * cAU)
    vst = (obVel target) - (obVel source)
    grcor' = ((4*ms/(r2*r2)) `scalarMult` rst) + ((4*dot rst vst/r3)
    ↪  `scalarMult` vst)

pNGRFull
  :: (Ord f, Floating f, Traversable t)
  => t (OrbitBody f)
  -> OrbitBody f
  -> CutParams f
  -> Vec3 f
pNGRFull bodies targ grcut = foldl (\acc b' -> acc + pNGRPair b' targ grcut) 0
↪  bodies
```

Listing 1: Physics.hs

```haskell
{-|
Module      : PhysicsVectors
Description : Implementation of two and three dimensional vectors as used in
↪  Physics

-}

{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}
```

```haskell
module PhysicsVectors
       (
         Vec(..)
       , Vec3(..)
       , Vec2(..)
       ) where

import GHC.Generics (Generic)
import Control.DeepSeq(NFData)


{-|
  The Vec class represents objects embedable within a normed vector space.

  Implementing Vec requires a concept of a dot product and scalar
↪  multiplication
  with the restriction that the embedded vector space be over a ring with
  a multiplicative inverse.

  More formally, instances should satisfy the three dot product rules, with the
  'fromScalar' requirement being an explicit representation of the implicit
  scalar multiplication process through a hadamard product.

  * \( x \cdot y = \overline{y \cdot x} \)
  * \( ax \cdot y = a(x \cdot y)\) and \((x + y) \cdot z = x \cdot z + x \cdot
↪  y \)
  * \( x \cdot x \geq 0 \)


-}
class (Num a, Floating b, Eq b) => Vec a b | a -> b where
  -- | 'fromScalar' @s@ returns a unit vector scaled by @s@
  fromScalar :: b -> a
  {-| 'dot' @v w@ returns the dot product of @v@ and @w@ as defined by the
     embedded vector space
  -}
  dot :: a -> a -> b
  normSq :: a -> b
  normSq w = dot w w
  norm :: a -> b
  norm = sqrt . normSq
  dist :: a -> a -> b
  dist w w' = norm $ w - w'
  unitVec :: a -> a
```

```
  unitVec w = (1/norm w) `scalarMult` w
  scalarMult :: b -> a -> a
  scalarMult 0 _ = fromScalar 0
  scalarMult s w = fromScalar s * w


{-|
  'Vec3' @x y z@ constructs a three dimensional cartesian vector with
  coordinates as noted.
-}
data Vec3 a = Vec3 { x::a, y::a, z::a }
  deriving (Eq, Ord, Show, Generic, NFData)

{-|
  'Vec2' @v1 v2@ constructs a two dimensional column vector of two three
  dimensional vectors of type 'Vec3'
-}
data Vec2 a = Vec2 { r :: Vec3 a, v :: Vec3 a }
  deriving (Eq, Ord, Show, Generic, NFData)

instance (Eq a, Floating a) => Vec (Vec3 a) a where
  fromScalar s = Vec3 s s s
  dot a b = sum [ q a * q b | q <- [x, y, z]]

instance (Eq a, Floating a) => Vec (Vec2 a) a where
  fromScalar s = Vec2 (fromScalar s) (fromScalar s)
  dot a b = sum [ dot (q a) (q b) | q <- [r, v]]

instance Floating a => Num (Vec3 a) where
  (Vec3 x1 y1 z1) + (Vec3 x2 y2 z2) = Vec3 (x1 + x2) (y1 + y2) (z1 + z2)
  (Vec3 x1 y1 z1) * (Vec3 x2 y2 z2) = Vec3 (x1 * x2) (y1 * y2) (z1 * z2)
  (Vec3 x1 y1 z1) - (Vec3 x2 y2 z2) = Vec3 (x1 - x2) (y1 - y2) (z1 - z2)
  abs (Vec3 x1 y1 z1) = Vec3 (abs x1) (abs y1) (abs z1)
  signum (Vec3 x1 y1 z1) = Vec3 (signum x1) (signum y1) (signum z1)
  fromInteger i = Vec3 (fromInteger i) (fromInteger i) (fromInteger i)

instance Floating a => Num (Vec2 a) where
  (Vec2 r1 v1) + (Vec2 r2 v2) = Vec2 (r1 + r2) (v1 + v2)
  (Vec2 r1 v1) * (Vec2 r2 v2) = Vec2 (r1 * r2) (v1 * v2)
  (Vec2 r1 v1) - (Vec2 r2 v2) = Vec2 (r1 - r2) (v1 - v2)
  abs (Vec2 r1 v1) = Vec2 (abs r1) (abs v1)
  signum (Vec2 r1 v1) = Vec2 (signum r1) (signum v1)
  fromInteger i = Vec2 (fromInteger i) (fromInteger i)
```

Listing 2: PhysicsVectors.hs

```haskell
{-|
Module      : AstroData
Description : Initial starting astronomy data

-}

{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}

module AstroData
       (
         OrbitBody(..)
       , obPos
       , obVel
       , updateSvecs
       , formatDoubleOB
       ) where

import PhysicsVectors
import Text.Printf(printf)
import GHC.Generics (Generic)
import Control.DeepSeq(NFData)

-- | Orbital body type storing the Body ID, GM parameter, position-velocity
 ↪  vector, and epoch
-- Note that units are in au days and epoch is normally measured from Julian --
 ↪  epoch
data OrbitBody f = OrbitBody { bid :: String, bGM :: f, brv :: Vec2 f, epoch ::
 ↪  f}
  deriving (Eq, Show, Generic, NFData)

obPos :: Floating f => OrbitBody f -> Vec3 f
obPos = r . brv

obVel :: Floating f => OrbitBody f -> Vec3 f
obVel = v . brv

updateSvecs :: Floating f => OrbitBody f -> f -> Vec2 f -> OrbitBody f
updateSvecs body dt rvn = OrbitBody (bid body) (bGM body) rvn (epoch body + dt)

formatDoubleOB :: OrbitBody Double -> String
formatDoubleOB (OrbitBody bid' _ (Vec2 r' v') t) = printf "%v, %v, %v, %v" bid'
 ↪  t (formatVec r') (formatVec v')
```

```
  where
    formatVec :: Vec3 Double -> String
    formatVec (Vec3 a b c) = printf "%v, %v, %v" a b c
```

Listing 3: AstroData.hs

```
{-|
Module      : PhysicsConstants
Description : Provides useful physics constants

Note that all SI /defined/ quantities are rational numbers as they are
/precisely/ defined as such. Any experimentally derived constants are provided
as doubles in as much precision as is known modulo floating point errors.

-}

module PhysicsConstants
       ( -- * Constants
         nGSI
       , cSI
       , cAU
       , au
       , aukm
       , secondsInDay
       , aGMtoSIGM
       ) where

-- | Speed of light in SI units (m s^-1)
cSI :: Rational -- Maaaaybe will need to use FixedPoint for everything
cSI = 299792458

-- | Speed of light in au units (au days^-1)
cAU :: Rational
cAU = cSI/au * secondsInDay

-- | Newton's Gravitational constant in SI units (m^3 kg^-1 s^-2), usually
↪  unused.
nGSI :: Double
nGSI = 6.674e-11

-- | Standard Astronomical Unit (au) as defined by IAU2012 in metres
au :: Rational
au = 1.495978707e11
```

```haskell
-- | Standard Astronomical Unit (au) as defined by IAU2012 in kilometres
aukm :: Rational
aukm = 1.495978707e8

-- | SI Defined number of seconds in non-standard unit day
secondsInDay :: Rational
secondsInDay = 86400

-- | Converts Astronomical GM (au^3 days^-2) to SIkm GM (km^3 s^-2)
aGMtoSIGM :: Floating f => f -> f
aGMtoSIGM gm = fromRational ((aukm^^(3::Int))/(secondsInDay^^(2::Int))) * gm
```

Listing 4: PhysicsConstants.hs

```haskell
{-|
Module      : Integrators
Description : Implementations of Runge-Kutta 8 and Gauss-Jackson integrators

Implements fourth, fifth, and eighth order Runge-Kutta integration methods for
time independent second order differential equations using the classic fourth
order, Dormand-Prince fifth order, and Cooper-Verner eigth order coefficients
respectively.
The Gauss-Jackson coefficients are generated to arbitrary order, but using
↪  RKCV8
for the startup conditions necessarily restricts the order to eigth order at
most, though correction steps can more or less minimise that loss. For some
inputs RK4 instead of RKCV8 or RKDP5 may make more sense for startup.

-}

module Integrators
       (
         rk4
       , rkdp5
       , rkcv8
       , timeIndepRK
       , Integrator(..)
       ) where

import PhysicsVectors
```

```haskell
-- | Time independent Butcher Tableu with a_ij in backwards row order
data BTabTIndep n = BTabTIndep [[n]] [n]
  deriving (Show)

rkcv8t :: Floating t => BTabTIndep t
rkcv8t = BTabTIndep
  [
    [1/2]
  , [1/4, 1/4]
  , [(21 + 5*s21)/49, (-7 - 3*s21)/98, 1/7]
  , [(21 - s21)/252, (18 + 4*s21)/63, 0, (11 + s21)/84]
  , [(63 - 7*s21)/80, (-231 + 14*s21)/360, (9 + s21)/36, 0, (5 + s21)/48]
  , [(63 - 13*s21)/35, (-504 + 115*s21)/70, (633 - 145*s21)/90
    , (-432 + 92*s21)/315, 0, (10 - s21)/42]
  , [1/9, (13 - 3*s21)/63, (14 - 3*s21)/126, 0, 0, 0, 1/14]
  , [(63 + 13*s21)/128, (-385 - 75*s21)/1152, 11/72, (91 - 21*s21)/576, 0, 0, 0,
  ↪  1/32]
  , [(132 + 28*s21)/245, (-51 - 11*s21)/56, (515 + 111*s21)/504
    , (-733 - 147*s21)/2205, 1/9, 0, 0, 0, 1/14]
  , [(49 - 7*s21)/18, (28 - 28*s21)/45, (301 + 53*s21)/72, (-273 - 53*s21)/72
    , (-18 + 28*s21)/45, (-42 + 7*s21)/18, 0, 0, 0, 0]
  ]
  [ 1/20, 49/180, 16/45, 49/180, 0, 0, 0, 0, 0, 0, 1/20]
  where
    s21 = sqrt 21

rk4t :: Floating t => BTabTIndep t
rk4t = BTabTIndep [[1/2],[1/2,0],[1,0,0]] [1/6,1/3,1/3,1/6]

rkdp5t :: Floating t => BTabTIndep t
rkdp5t = BTabTIndep
  [
    [1/5]
  , [9/40, 3/40]
  , [32/9, -56/15, 44/45]
  , [-212/729, 64448/6561, -25360/2187, 19372/6561]
  , [-5103/18656, 49/176, 46732/5247, -355/33, 9017/3168]
  , [11/84, -2187/6784, 125/192, 500/1113, 0, 35/384]
  ]
  [0, 11/84, -2187/6784, 125/192, 500/1113, 0, 35/384]

{-|
  'timeIndepRK' implements the general Runge-Kutta form for a time independent
  differential equation \(f\).
```

```haskell
  'timeIndepRK' @btab f yn dt@ steps one step forward in the described
  time-independent Runge-Kutta method where @btab@ describes the implementation
  Butcher-Tableu, @f@ the differetial function, @rn@ the current position and
  velocity, and @dt@ the integration time step.

  Note that \(f\) operates on both position and velocity, i.e. it returns both
  \(\dot{\vec{r}}\) and \(\ddot{\vec{r}}\) and that the Butcher-Tableu's used
  must have the \(a_{ij}\) and \(b_i\) coefficients in /reverse/ row order.
↪  That
  is, each row is in reverse order.
-}
timeIndepRK
    :: (Floating t, Eq t)
    => BTabTIndep t
    -> (Vec2 t -> Vec2 t)
    -> Vec2 t
    -> t
    -> Vec2 t
timeIndepRK _ _ rn 0 = rn
timeIndepRK (BTabTIndep a b) f rn dt = step ks b
  where
    coeffSum [] _ = 0
    coeffSum kl zetas = sum [ zi `scalarMult` ki | (ki, zi) <- zip kl zetas ]
    ks = foldl (\kl aj-> f (step kl aj) : kl) [f rn] a
    step kl zi = rn + (dt `scalarMult` coeffSum kl zi)

-- | Data class encapsulating the concept of an integrator function
data Integrator t =
    SingStepIntegrator ((Vec2 t -> Vec2 t) -> Vec2 t -> t -> Vec2 t)
  | MultStepIntegrator ((Vec2 t -> Vec2 t) -> [(Vec2 t, Vec3 t)] -> t -> [(Vec2
  ↪  t, Vec3 t)])


-- | The classic Runge-Kutta fourth order method
rk4 :: (Floating t, Eq t) => Integrator t
rk4 = SingStepIntegrator $ timeIndepRK rk4t

{-|
  Dormand-Prince fifth order method. A less general RK method could avoid the
  final b calculation and the first f calculation of the next step with this
↪  one
  due to the coefficient choice.
-}
```

```haskell
rkdp5 :: (Floating t, Eq t) => Integrator t
rkdp5 = SingStepIntegrator $ timeIndepRK rkdp5t

-- | Cooper-Verner eigth order method
rkcv8 :: (Floating t, Eq t) => Integrator t
rkcv8 = SingStepIntegrator $ timeIndepRK rkcv8t
```

Listing 5: Integrators.hs

```haskell
module Main where

import Integrators
import Physics
import AstroData
--import AsteroidData
--import PlanetData
import AllBodiesData

import System.IO(IOMode(WriteMode), withFile, stderr, hPutStrLn)
import System.Environment(getArgs, getProgName)
import System.Exit(exitFailure)

import Control.Parallel.Strategies(using, rdeepseq, parTraversable)
import Control.DeepSeq(NFData)

{-|
  Default GR cutoff values such that Jupiter is massive enough and Mercury
  close enough for GR effects
-}
defGrcut :: Floating t => CutParams t
defGrcut = CutParams 0.2e-6 0.5

main :: IO ()
main = do args <- getArgs
          case args of
            [steps, dt, filename] -> do
              let (steps', dt') = (1 + (read steps :: Int), read dt :: Double)
              --let ephem = generateEphemerides rkcv8 (majorbodies ++
              ↪  asteroids) defGrcut dt' steps'
              let ephem = generateEphemeridesPar rkcv8 allbodies defGrcut dt'
              ↪  steps'
              withFile filename WriteMode (\h -> mapM_ (\ob->mapM_ ((hPutStrLn
              ↪  h). formatDoubleOB) ob) ephem)
```

```
            _ -> do pn <- getProgName
                    hPutStrLn stderr $ "Usage: " ++ pn ++ " steps dt filename"
                    exitFailure


integrableForceEq :: (Floating f, Ord f) => [ OrbitBody f ] -> CutParams f ->
↪  OrbitBody f -> Vec2 f -> Vec2 f
integrableForceEq bodies  grcut (OrbitBody bid' bGM' _ et') svec =
  Vec2 (v svec) $ pNGRFull bodies (OrbitBody bid' bGM' svec et') grcut

stepForwardAll :: (Floating t, Eq t) => Integrator t -> (OrbitBody t -> Vec2 t
↪  -> Vec2 t) -> t -> [OrbitBody t] -> [OrbitBody t]
stepForwardAll (SingStepIntegrator i) f dt bodies =
  [ updateSvecs body dt $ i (f body) (brv body) dt | body <- bodies ]
stepForwardAll _ _ _ _ = undefined


stepForwardAllPar :: (Floating t, Eq t, NFData t) => Integrator t -> (OrbitBody
↪  t -> Vec2 t -> Vec2 t) -> t -> [ OrbitBody t ] -> [ OrbitBody t ]
stepForwardAllPar (SingStepIntegrator i) f dt bodies =
  fmap method bodies `using` parTraversable rdeepseq
  where
    method body = updateSvecs body dt $ i (f body) (brv body) dt
stepForwardAllPar _ _ _ _ = undefined

  {-
    -- Silly explicit version of stepForwardAllPar that shows that my current
    -- form is in fact not working, only applying to two cores whereas this
    -- version does succesfully work on three cores. It also sparks way too
↪   many
    -- and fails to be any faster.
stepForwardAllPar :: (Floating t, Eq t, NFData t) => Integrator t -> (OrbitBody
↪   t -> Vec2 t -> Vec2 t) -> t -> [ OrbitBody t ] -> [ OrbitBody t ]
stepForwardAllPar (SingStepIntegrator i) f dt bodies =
  --fmap method bodies `using` parTraversable rdeepseq
  chunk3 method bodies
  where
    method body = updateSvecs body dt $ i (f body) (brv body) dt
    chunk3 fn (l1:l2:l3:ls) = (t3tol $ runEval (parTuple3 rdeepseq rdeepseq
↪   rdeepseq (fn l1, fn l2, fn l3))) ++ (chunk3 fn ls)
    chunk3 fn l = map fn l `using` parList rdeepseq
    t3tol (l1, l2, l3) = [l1, l2, l3]
stepForwardAllPar _ _ _ _ = undefined
-}
```

```haskell
generateEphemeridesPar
  :: (NFData t, Floating t, Ord t)
  => Integrator t
  -> [ OrbitBody t ]
  -> CutParams t
  -> t
  -> Int
  -> [ [ OrbitBody t ]]
generateEphemeridesPar integrator bodies grcut dt steps =
  take steps (iterate advance bodies)
  where
    advance bodies' = stepForwardAllPar integrator (f bodies') dt bodies'
    f bs' = integrableForceEq bs' grcut


generateEphemerides
  :: (Floating t, Ord t)
  => Integrator t
  -> [OrbitBody t]
  -> CutParams t
  -> t
  -> Int
  -> [[OrbitBody t]]
generateEphemerides integrator bodies grcut dt steps =
  take steps (iterate advance bodies)
  where
    advance bodies' = stepForwardAll integrator (f bodies') dt bodies'
    f bs' = integrableForceEq bs' grcut
--generateEphemerides (SingStepIntegrator i) bodies grcut dt steps = undefined
--  --take steps (i (integrableForceEq ))
--generateEphemerides (MultStepIntegrator i) bodies grcut dt steps = undefined
```

Listing 6: Main.hs

All "Data" files are omitted as they are literally bare bones modules designed to store the starting data set, see "AstroData.hs" for the general format.