```
1   using System;
2   using System.Collections.Generic;
3   using static Program.Constants;
4   using System.Threading;
5   using System.Threading.Tasks;
6   using System.Linq;
7   namespace Structures
8   {
9           [Serializable()]
10          public class Vector3 {
11                  // Simple 3-vector class, used for positions, velocities,
    color, etc.
12                  // setters are required for deserialization but should not be
    used outside class
13                  public double x {get; set;}
14                  public double y {get; set;}
15                  public double z {get; set;}
16                  public Vector3() {} // paramaterless constructor for
    serialization
17                  public Vector3(double x, double y, double z) {
18                          this.x = x;
19                          this.y = y;
20                          this.z = z;
21                  }
22                  // Immutable standard vectors
23                  public static Vector3 zero {get;} = new Vector3(0,0,0);
24                  public static Vector3 i {get;} = new Vector3(1,0,0);
25                  public static Vector3 j {get;} = new Vector3(0,1,0);
26                  public static Vector3 k {get;} = new Vector3(0,0,1);
27                  public override String ToString() {
28                          return $"Vector3({x},{y},{z})";
29                  }
30                  public static bool operator== (Vector3 a, Vector3 b) {
31                          // why reimplement null checks ourselves?
32                          if ((object)a == null || ((object)b == null)) return
    (object)a == null && (object)b == null;
33                          // otherwise return true if all components are within
    10^-10
34                          bool[] eq = new bool[3];
35                          for (int i = 0; i < 3; i++) {
36                                  double a1,b1;
37                                  if (i == 0) {a1 = a.x; b1 = b.x;}
38                                  else if (i == 1) {a1 = a.y; b1 = b.y;}
39                                  else {a1 = a.z; b1 = b.z;}
40                                  if (Math.Abs(a1) < 1e-2 || Math.Abs(b1) <
    1e-2) {
41                                          eq[i] = Math.Abs(a1 - b1) < 1e-10;
42                                  } else {
43                                          eq[i] = Math.Abs((a1-b1)/a1) < 1e-10
44                                               && Math.Abs((a1-b1)/b1) < 1e-10;
45                                  }
46                          }
47                          return eq[0] && eq[1] && eq[2];
48                  }
49                  public static bool operator!= (Vector3 a, Vector3 b) {
50                          // inverse of equality operator
51                          return !(a == b);
52                  }
53                  public static Vector3 operator- (Vector3 a, Vector3 b) {
54                          return new Vector3 (a.x-b.x,a.y-b.y,a.z-b.z);
55                  }
56                  public static Vector3 operator- (Vector3 a) {
57                          return new Vector3(-a.x,-a.y,-a.z);
58                  }
59                  public static Vector3 operator+ (Vector3 a, Vector3 b) {
60                          return new Vector3 (a.x+b.x,a.y+b.y,a.z+b.z);
```

```
61                  }
62                  public static Vector3 operator* (double a, Vector3 b) {
63                          return new Vector3 (a*b.x,a*b.y,a*b.z);
64                  }
65                  public static Vector3 operator/ (Vector3 a, double b) {
66                          return new Vector3 (a.x/b,a.y/b,a.z/b);
67                  }
68                  public static double dot(Vector3 a, Vector3 b) {
69                          // This could be overloaded to operator*, but an
    explicit function increases readibility.
70                          return a.x*b.x + a.y*b.y + a.z*b.z;
71                  }
72                  public static Vector3 cross(Vector3 a, Vector3 b) {
73                          return new Vector3(
74                                  a.y*b.z - a.z*b.y,
75                                  a.z*b.x - a.x*b.z,
76                                  a.x*b.y - a.y*b.x
77                          );
78                  }
79                  public static double Magnitude(Vector3 v) {
80                          // Pythagorean Theorem
81                          return Math.Sqrt(Math.Pow(v.x,2)+Math.Pow(v.y,2)
    +Math.Pow(v.z,2));
82                  }
83                  public static Vector3 Unit(Vector3 v) {
84                          if (v == Vector3.zero) {
85                                  throw new DivideByZeroException("Cannot take
    unit of zero vector");
86                          }
87                          return v / Vector3.Magnitude(v);
88                  }
89                  public static double UnitDot(Vector3 a, Vector3 b) {
90                          // The dot of the unit vectors
91                          return Vector3.dot(Vector3.Unit(a),Vector3.Unit(b));
92                  }
93                  public static Vector3 Log(Vector3 v, double b = Math.E) {
94                          // Polar logarithm (radius is logged, direction is
    consistent)
95                          var polar = CartesianToPolar(v);
96                          var log_polar = new Vector3 (Math.Log
    (polar.x,b),polar.y,polar.z);
97                          var log = PolarToCartesian(log_polar);
98                          return log;
99                  }
100                 public static Vector3 LogByComponent(Vector3 v, double b =
    Math.E) {
101                         // Cartesian Logarithm, all components are logged
102                         var r = new Vector3(0,0,0);
103                         // using Vector3.zero will modify it, since we are
    inside the Vector class,
104                         // where Vector3.zero is mutable
105                         if (v.x < 0) r.x = -Math.Log(-v.x,b);
106                         else if (v.x != 0) r.x = Math.Log(v.x,b);
107                         if (v.y < 0) r.y = -Math.Log(-v.y,b);
108                         else if (v.y != 0) r.y = Math.Log(v.y,b);
109                         if (v.z < 0) r.z = -Math.Log(-v.z,b);
110                         else if (v.z != 0) r.z = Math.Log(v.z,b);
111                         return r;
112                 }
113                 public static Vector3 CartesianToPolar(Vector3 v) {
114                         // ISO Convention
115                         var r = Vector3.Magnitude(v);
116                         var theta = Math.Acos(Vector3.UnitDot(v,Vector3.k));
117                         var phi = Math.Acos(Vector3.UnitDot(new Vector3
    (v.x,v.y,0),Vector3.i));
118                         if (v.y < 0) phi = -phi;
```

```
119                         return new Vector3(r,theta,phi);
120                 }
121                 public static Vector3 PolarToCartesian(Vector3 v) {
122                         // ISO Convention
123                         return Matrix3.ZRotation(v.z) * Matrix3.YRotation
    (v.y) * (v.x*Vector3.k);
124                 }
125
126         }
127         public class Matrix3 {
128                 // the fields describe the rows. Using Vector3s makes Matrix-
    Vector Multiplication
129                 // (which is the most useful operation) simpler, since then
    Vector3.dot can be used
130                 public Vector3 x {get;}
131                 public Vector3 y {get;}
132                 public Vector3 z {get;}
133                 public Matrix3(Vector3 x, Vector3 y, Vector3 z) {
134                         this.x = x;
135                         this.y = y;
136                         this.z = z;
137                 }
138                 public override String ToString() {
139                         return $"Matrix3( {x.x} {x.y} {x.z}\n         {y.x}
    {y.y} {y.z}\n         {z.x} {z.y} {z.z} )";
140                 }
141                 public static Matrix3 XRotation(double x) {
142                         return new Matrix3 (
143                                 new Vector3(1,0,0),
144                                 new Vector3(0,Math.Cos(x),Math.Sin(x)),
145                                 new Vector3(0,-Math.Sin(x),Math.Cos(x))
146                         );
147                 }
148                 public static Matrix3 YRotation(double y) {
149                         return new Matrix3 (
150                                 new Vector3(Math.Cos(y),0,Math.Sin(y)),
151                                 new Vector3(0,1,0),
152                                 new Vector3(-Math.Sin(y),0,Math.Cos(y))
153                         );
154                 }
155                 public static Matrix3 ZRotation(double z) {
156                         return new Matrix3 (
157                                 new Vector3(Math.Cos(z),-Math.Sin(z),0),
158                                 new Vector3(Math.Sin(z),Math.Cos(z),0),
159                                 new Vector3(0,0,1)
160                         );
161                 }
162                 public static Matrix3 ExtrinsicZYXRotation(double x, double
    y, double z) {
163                         return XRotation(x)*YRotation(y)*ZRotation(z);
164                 }
165                 public static Matrix3 ExtrinsicZYXRotation(Vector3 v) {
166                         return XRotation(v.x)*YRotation(v.y)*ZRotation(v.z);
167                 }
168                 public static Matrix3 IntrinsicZYXRotation(double x, double
    y, double z) {
169                         return ZRotation(z)*YRotation(y)*XRotation(x);
170                 }
171                 public static Matrix3 IntrinsicZYXRotation(Vector3 v) {
172                         return ZRotation(v.z)*YRotation(v.y)*XRotation(v.x);
173                 }
174                 public static bool operator== (Matrix3 a, Matrix3 b) {
175                         return a.x == b.x && a.y == b.y && a.z == b.z;
176                 }
177                 public static bool operator!= (Matrix3 a, Matrix3 b) {
178                         return !(a == b);
```

```
179                 }
180
181                 public static Matrix3 operator+ (Matrix3 a, Matrix3 b) {
182                         // Add component-wise
183                         return new Matrix3(
184                                 a.x + b.x,
185                                 a.y + b.y,
186                                 a.z + b.z
187                         );
188                 }
189                 public static Vector3 operator* (Matrix3 m, Vector3 v) {
190                         return new Vector3(
191                                 Vector3.dot(m.x,v),
192                                 Vector3.dot(m.y,v),
193                                 Vector3.dot(m.z,v)
194                         );
195                 }
196                 public static Matrix3 operator* (double d, Matrix3 m) {
197                         // multiply each component by d
198                         return new Matrix3(
199                                 d * m.x,
200                                 d * m.y,
201                                 d * m.z
202                         );
203                 }
204                 public static Matrix3 operator/ (Matrix3 m, double d) {
205                         if (d == 0) throw new DivideByZeroException("Matrix
    Division By Zero");
206                         else return (1/d) * m;
207                 }
208                 public static Matrix3 operator* (Matrix3 l, Matrix3 r) {
209                         var r_t = Matrix3.Transpose(r);
210                         return new Matrix3 (
211                                 new Vector3(
212                                         Vector3.dot(l.x,r_t.x),
213                                         Vector3.dot(l.x,r_t.y),
214                                         Vector3.dot(l.x,r_t.z)
215                                 ),
216                                 new Vector3(
217                                         Vector3.dot(l.y,r_t.x),
218                                         Vector3.dot(l.y,r_t.y),
219                                         Vector3.dot(l.y,r_t.z)
220                                 ),
221                                 new Vector3(
222                                         Vector3.dot(l.z,r_t.x),
223                                         Vector3.dot(l.z,r_t.y),
224                                         Vector3.dot(l.z,r_t.z)
225                                 )
226                         );
227                 }
228                 public static double Determinant(Matrix3 m) {
229                         return m.x.x * (m.y.y*m.z.z - m.y.z*m.z.y)
230                               -m.x.y * (m.y.x*m.z.z - m.y.z*m.z.x)
231                                  +m.x.z * (m.y.x*m.z.y - m.y.y*m.z.x);
232                 }
233                 public static Matrix3 Transpose(Matrix3 m) {
234                         return new Matrix3(
235                                 new Vector3(m.x.x,m.y.x,m.z.x),
236                                 new Vector3(m.x.y,m.y.y,m.z.y),
237                                 new Vector3(m.x.z,m.y.z,m.z.z)
238                         );
239                 }
240                 public static Matrix3 TransposeCofactor(Matrix3 m) {
241                         // We never need to do the cofactor without the
    transpose, so this is an optimisation
242                         return new Matrix3(
```

```
243                                 new Vector3(m.x.x,-m.y.x,m.z.x),
244                                 new Vector3(-m.x.y,m.y.y,-m.z.y),
245                                 new Vector3(m.x.z,-m.y.z,m.z.z)
246                         );
247                 }
248                 public static Matrix3 Minor(Matrix3 m) {
249                         return new Matrix3(
250                                 new Vector3(
251                                         (m.y.y*m.z.z - m.y.z*m.z.y),
252                                         (m.y.x*m.z.z - m.y.z*m.z.x),
253                                         (m.y.x*m.z.y - m.y.y*m.z.x)
254                                 ),
255                                 new Vector3(
256                                         (m.x.y*m.z.z - m.x.z*m.z.y),
257                                         (m.x.x*m.z.z - m.x.z*m.z.x),
258                                         (m.x.x*m.z.y - m.x.y*m.z.x)
259                                 ),
260                                 new Vector3(
261                                         (m.x.y*m.y.z - m.x.z*m.y.y),
262                                         (m.x.x*m.y.z - m.x.z*m.y.x),
263                                         (m.x.x*m.y.y - m.x.y*m.y.x)
264                                 )
265                         );
266                 }
267                 public static Matrix3 Inverse(Matrix3 m) {
268                         if (Matrix3.Determinant(m) == 0) throw new
    DivideByZeroException("Singular Matrix");
269                         Matrix3 C_T = Matrix3.TransposeCofactor(Matrix3.Minor
    (m));
270                         return (1/Matrix3.Determinant(m)) * C_T;
271                 }
272         }
273         [Serializable()]
274         public class Body : ICloneable {
275                 public string name {get; set;}
276                 public Body parent {get; set;}
277                 // standard gravitational parameter
278                 public double stdGrav {get; set;}
279                 public double radius {get; set;}
280                 public Vector3 position {get; set;} = Vector3.zero;
281                 public Vector3 velocity {get; set;} = Vector3.zero;
282                 public Vector3 color {get; set;} = new Vector3(1,1,1);
283                 public Body() {} // paramaterless constructor for
    serialisation
284                 public Body (Body parent, OrbitalElements elements) {
285                         // First check the values are reasonable. If parent
    == null it is assumed that
286                         // position and velocity are set explicitly, and this
    constructor is not used
287                         if (parent == null) return;
288                         this.parent = parent;
289                         if (elements.eccentricity < 0
290                          || elements.semilatusrectum < 0
291                          || elements.inclination < 0
292                          || elements.inclination > Math.PI
293                          || elements.ascendingNodeLongitude < 0
294                          || elements.ascendingNodeLongitude >= 2*Math.PI
295                          || elements.periapsisArgument < 0
296                          || elements.periapsisArgument >= 2*Math.PI
297                          || elements.trueAnomaly < 0
298                          || elements.trueAnomaly >= 2*Math.PI
299                         ){
300                                 // Throw an exception if the arguments are
    out of bounds
301                                 throw new ArgumentException();
302                         }
```

```
303                         double semilatusrectum = elements.semilatusrectum;//*
    (1-Math.Pow(elements.eccentricity,2));
304                         // working in perifocal coordinates (periapsis along
    the x axis, orbit in the x,y plane):
305                         double mag_peri_radius = semilatusrectum/(1
    +elements.eccentricity*Math.Cos(elements.trueAnomaly));
306                         Vector3 peri_radius = mag_peri_radius*new Vector3
    (Math.Cos(elements.trueAnomaly),Math.Sin(elements.trueAnomaly),0);
307                         Vector3 peri_velocity = Math.Sqrt(parent.stdGrav/
    semilatusrectum)
308                                                                         * new
    Vector3(
309                                                                             -
    Math.Sin(elements.trueAnomaly),
310
    Math.Cos(elements.trueAnomaly) + elements.eccentricity,
311
    0
312                                                                         );
313                         // useful constants to setup matrix
314                         var sini = Math.Sin(elements.inclination); // i ->
    inclination
315                         var cosi = Math.Cos(elements.inclination);
316                         var sino = Math.Sin
    (elements.ascendingNodeLongitude); // capital omega -> longitude of ascending
    node
317                         var coso = Math.Cos(elements.ascendingNodeLongitude);
318                         var sinw = Math.Sin(elements.periapsisArgument); //
    omega -> argument of periapsis
319                         var cosw = Math.Cos(elements.periapsisArgument);
320                         // As described by the book "Fundamentals of
    Astrodynamics",
321                         // we transform perifocal coordinates to i,j,k
    coordinates
322                         Matrix3 transform = new Matrix3(
323                                 new Vector3(
324                                         coso*cosw - sino*sinw*cosi,
325                                         -coso*sinw-sino*cosw*cosi,
326                                         sino*sini
327                                 ),
328                                 new Vector3(
329                                         sino*cosw+coso*sinw*cosi,
330                                         -sino*sinw+coso*cosw*cosi,
331                                         -coso*sini
332                                 ),
333                                 new Vector3(
334                                         sinw*sini,
335                                         cosw*sini,
336                                         cosi
337                                 )
338                         );
339                         // add the parent's position and velocity since that
    could be orbiting something too
340                         this.position = transform*peri_radius +
    parent.position;
341                         this.velocity = transform*peri_velocity +
    parent.velocity;
342                 }
343                 public double HillRadius() {
344                         // This is the maximum distance anything can
    reasonably orbit at.
345                         // It would normally depend on the bodies nearby, but
    we'll just do something simple
346                         // which is roughly accurate for bodies in the solar
    system.
347                         return this.stdGrav * 1e-6;
```

```
348                 }
349                 public object Clone() {
350                         return new Body {
351                                 name = this.name,
352                                 parent = this.parent,
353                                 stdGrav = this.stdGrav,
354                                 radius = this.radius,
355                                 position = this.position,
356                                 velocity = this.velocity,
357                                 color = this.color
358                         };
359                 }
360         }
361         internal class FundamentalVectors {
362                 // The fundamental vectors of an orbit
363                 public Vector3 angularMomentum {get; set;}
364                 public Vector3 eccentricity {get; set;}
365                 public Vector3 node {get; set;}
366                 public FundamentalVectors(Vector3 position, Vector3 velocity,
    double stdGrav) {
367                         this.angularMomentum = Vector3.cross
    (position,velocity);
368                         this.node = Vector3.cross
    (Vector3.k,this.angularMomentum);
369                         var mag_r = Vector3.Magnitude(position);
370                         var mag_v = Vector3.Magnitude(velocity);
371                         this.eccentricity = (1/stdGrav)*((Math.Pow(mag_v,2) -
    stdGrav/mag_r)*position - Vector3.dot(position,velocity)*velocity);
372                 }
373                 public override String ToString() {
374                         return $"Angular Momentum: {angularMomentum.ToString
    ()}\nEccentricity: {eccentricity.ToString()}\nNode: {node.ToString()}";
375                 }
376
377         }
378         public class OrbitalElements {
379                 // The six classical orbital elements
380                 public double semilatusrectum {get; set;}
381                 public double eccentricity {get; set;}
382                 protected double _inclination;
383                 public double inclination {
384                         get {
385                                 return _inclination;
386                         } set {
387                                 _inclination = value%Math.PI;
388                         }
389                 }
390                 protected double _ascendingNodeLongitude;
391                 public double ascendingNodeLongitude {
392                         get {
393                                 return _ascendingNodeLongitude;
394                         } set {
395                                 _ascendingNodeLongitude = value%(2*Math.PI);
396                         }
397                 }
398                 protected double _periapsisArgument;
399                 public double periapsisArgument {
400                         get {
401                                 return _periapsisArgument;
402                         } set {
403                                 _periapsisArgument = value%(2*Math.PI);
404                         }
405                 }
406                 protected double _trueAnomaly;
407                 public double trueAnomaly {
408                         get {
```

```
409                                     return _trueAnomaly;
410                             } set {
411                                     _trueAnomaly = value%(2*Math.PI);
412                             }
413                     }
414                     public OrbitalElements() {} // For serialisation
415                     public OrbitalElements(Vector3 position, Vector3 velocity,
    double stdGrav) {
416                             // stdGrav is the gravitational parameter of the
    parent body
417                             var fVectors = new FundamentalVectors
    (position,velocity,stdGrav);
418                             this.eccentricity = Vector3.Magnitude
    (fVectors.eccentricity);
419                             var semilatusrectum = Math.Pow(Vector3.Magnitude
    (fVectors.angularMomentum),2)/stdGrav;
420                             this.semilatusrectum = semilatusrectum;///(1-Math.Pow
    (eccentricity,2));
421                             this.inclination = Math.Acos
    (fVectors.angularMomentum.z/Vector3.Magnitude(fVectors.angularMomentum)); //
    0 <= i <= 180deg
422                             //TODO: fix parabola
423
424                             double cosAscNodeLong = fVectors.node.x/
    Vector3.Magnitude(fVectors.node);
425                             if (fVectors.node.y >= 0) this.ascendingNodeLongitude
    = Math.Acos(cosAscNodeLong);
426                             else this.ascendingNodeLongitude = 2*Math.PI -
    Math.Acos(cosAscNodeLong);
427                             double cosAnomaly = 0;
428                             try {
429                                     double cosPeriArg = Vector3.UnitDot
    (fVectors.node,fVectors.eccentricity);
430                                     if (fVectors.eccentricity.z >= 0)
    this.periapsisArgument = Math.Acos(cosPeriArg);
431                                     else this.periapsisArgument = 2*Math.PI -
    Math.Acos(cosPeriArg);
432                                     cosAnomaly = Vector3.UnitDot
    (fVectors.eccentricity,position);
433                             } catch (DivideByZeroException) {
434                                     // This will be dealt with along with
    extremely small values below
435                             }
436                             if (this.eccentricity < 1e-10 ) {
437                                     // acceptable error, the orbit has no
    periapsis
438                                     this.eccentricity = 0;
439                                     this.periapsisArgument = 0;
440                                     // we assume the periapsis is at the node
    vector
441                                     if (Vector3.Magnitude(fVectors.node) < 1e-10)
    {
442                                             // but if the node vector also does
    not exist we assume the i vector
443                                             cosAnomaly = Vector3.UnitDot
    (Vector3.i,position);
444                                     } else {
445                                             cosAnomaly = Vector3.UnitDot
    (fVectors.node, position);
446                                     }
447                             }
448                             if (Vector3.UnitDot(position,velocity) >= 0)
    this.trueAnomaly = Math.Acos(cosAnomaly);
449                             else this.trueAnomaly = 2*Math.PI - Math.Acos
    (cosAnomaly);
450                             if (Math.Abs(fVectors.angularMomentum.x/
```

```
      fVectors.angularMomentum.z) < 1e-10
451                            && Math.Abs(fVectors.angularMomentum.y/
      fVectors.angularMomentum.z) < 1e-10) {
452                                    // acceptable error, the orbit is not inclined
453                                    this.ascendingNodeLongitude = 0;
454                            }
455                            if (this.ascendingNodeLongitude >= 2*Math.PI)
      this.ascendingNodeLongitude -= 2*Math.PI;
456                            if (this.periapsisArgument >= 2*Math.PI)
      this.periapsisArgument -= 2*Math.PI;
457                            if (this.trueAnomaly >= 2*Math.PI) this.trueAnomaly -
      = 2*Math.PI;
458                    }
459            }
460    }
```