Don't go sick to Malta: How a disease would spread in the EU28 zone

Henrik Åhl
in collaboration with Denhanh Huynh

November 4, 2015

Department of Astronomy and Theoretical Physics, Lund University

Project supervised by Tobias Ambjörnsson

LUND UNIVERSITY

Introduction

How are diseases spread within Europe depending on how people travel? And perhaps more importantly: Where would the initial occurence of a spread end up affecting the most people? In this report we investigate effects on disease spread based on travel by flight between countries being part of the EU28 project. The two main purposes of this investigation is to discern the pattern of spread, and to investigate where the initial strike of a disease would cause the greatest impact.

We do in this report show that, given a range of simplified assumptions, it is possible to construct a model which in several regards produces results that roughly predict the spreading of a disease in the EU28 zone to a realistical degree.

Modelling how diseases spread

The spreading of the disease was simulated with the SIR model, categorizing people into three different categories: Susceptible, Infected and Recovered (Resistant). For a time step $\mathrm{d}t$, which in this report signifies the passing of a full week, it thus follows that

$$\frac{\mathrm{d}S(t)}{\mathrm{d}t} + \frac{\mathrm{d}I(t)}{\mathrm{d}t} + \frac{\mathrm{d}R(t)}{\mathrm{d}t} = 0 \tag{0.1}$$

and it must subsequently be the case that $S(t) + I(t) + R(t)$ is equal to a constant, which naturally is the population size. Because the population size has to remain constant throughout every time step in the SIR model, the number of people travelling between two countries had to be averaged to fulfill this requirement. The data by itself was acquired via the Eurostat database [1] and modified to fit the model adequatly. Our data presented is labelled country-wise according to each country's corresponding country code, e.g. SE for Sweden, and so forth.

From the SIR model, three first-order differential equations arise for a country n:

$$\frac{\mathrm{d}S_n}{\mathrm{d}t} = -\beta\frac{S_n I_n}{N_n} - \gamma S_n + \sum_{m\neq n}\left[\omega_{n\leftarrow m}S_m - \omega_{m\leftarrow n}S_n\right] \tag{0.2}$$

$$\frac{\mathrm{d}I_n}{\mathrm{d}t} = \beta\frac{S_n I_n}{N_n} - \alpha I_n + \sum_{m\neq n}\left[\omega_{n\leftarrow m}I_m - \omega_{m\leftarrow n}I_n\right] \tag{0.3}$$

$$\frac{\mathrm{d}R_n}{\mathrm{d}t} = \gamma S_n + \alpha I_n + \sum_{m\neq n}\left[\omega_{n\leftarrow m}R_m - \omega_{m\leftarrow n}R_n\right] \tag{0.4}$$

These were all calculated numerically using a fourth-order Runge-Kutta method, allowing for a local error scaling as the step size to the fourth power. In the SIS-model, the model is instead represented by the equations

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -\beta\frac{SI}{N} + \alpha I$$
$$\frac{\mathrm{d}I}{\mathrm{d}t} = -\beta\frac{SI}{N} - \alpha I$$

where one can note that an equilibrium state, i.e. where all derivatives are zero, will imply that the system will tend towards the fraction $f = I/N = 1 - \alpha/\beta$. This characteristic is however not inherent to our model, as the systems in question are not isolated.

Moreover, our α, β and γ represent the rate of recovery (with implied afterwardly immunity), the rate of spread due to the number of contacts

and the probability of transmission, and the vaccination rate of infected subjects respectively [3]. All parameters apply to the weekly impact of each characteristic of the system. The parameters controlling the flux of births and deaths within a population was neglected due to the low impact this would have on the dynamics for a disease bearing low lethality.

The weights ω represent the quota of the population of a country that travels in between the countries n and m. α, β and γ were for this report, unledd otherwise stated, set to values of 1, 2.5 and 0.002 respectively – mirroring one effective week for recovery, 2.5 transmittances per infected person and week, and 0.2 percent of the population being vaccinated each week. All values were chosen in pursuit to mimic a highly-transmittable, non-lethal disease, similiar to many rhinoviri, which typically cause about 10–40 % of all cold-outbreaks [2].

The reader should take note that with the requirement of $\mathrm{d}N/\,\mathrm{d}t = 0$ it is also necessarily the case that $\omega_{n\leftarrow m} S_n = \omega_{m\leftarrow n} S_m$.

In the modelling of the spread, all countries commence vaccinations at the very first occurence of the disease in amongst the EU28 countries, therefore allowing for countries that lie further away from the initial spread time-wise to begin with countermeasures before the disease has reached that actual country.

For the error calculation, an estimate was produced with the formula

$$E_n^{(h)} = \frac{y_n^{(h)} - y_n^{(2h)}}{2^m - 1} \tag{0.5}$$

where m signifies the order of the algorithm used for the calculations. In this case, since Runge-Kutta is of the fourth order, $m = 4$.

Results

The simulations provide results that agree with the prediction of delayed transmittance. When the disease is initially inserted into one individual in a country, other countries produce a delayed response according to the asserted travel rates. The error estimate in the total population size of each country was calculated to a value on the order of magnitude 10^0, which, since the average population size is on the order 10^7, implies a relative error of the order 10^{-7} per country.

There is no decisive pattern for how many people are ultimately affected by the disease, although there seems to be a slight bias towards countries with a high rate of international travel, such as MT, which in fig. 1 stands out as being the most dangerous country for an outbreak to begin in.

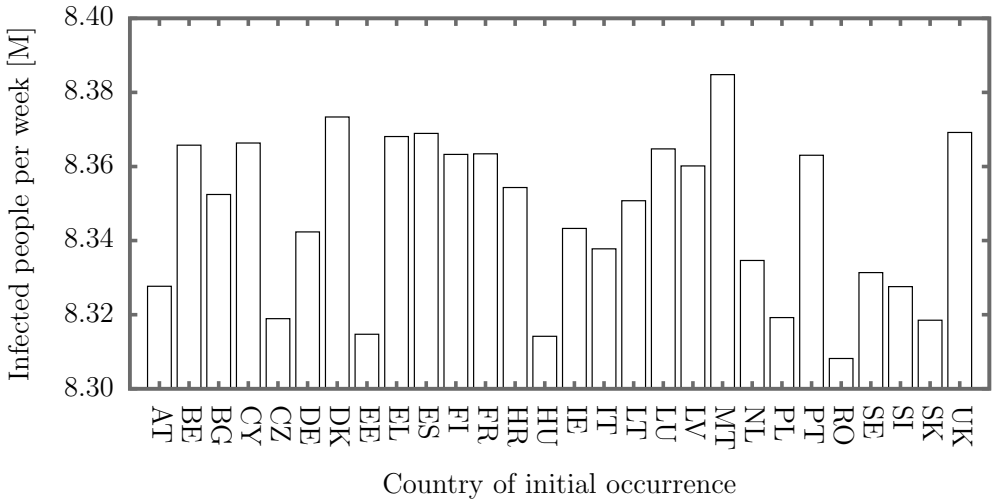

Figure 1: Average number of infected individuals across the whole of the EU28 zone, with data per week and starting country.

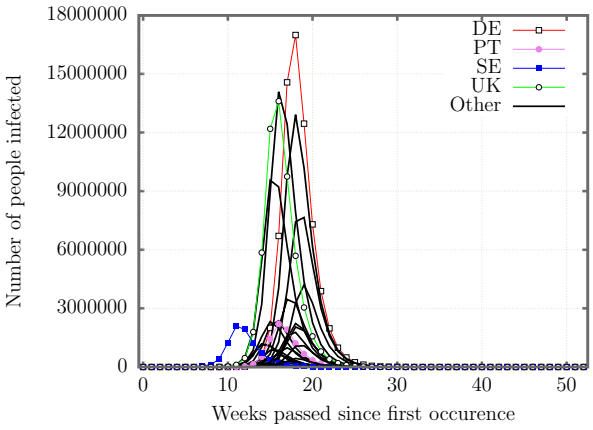

Figure 2: Infection dynamics with SE as initial target. Note the delayed effect for other countries with respect to reaction time.

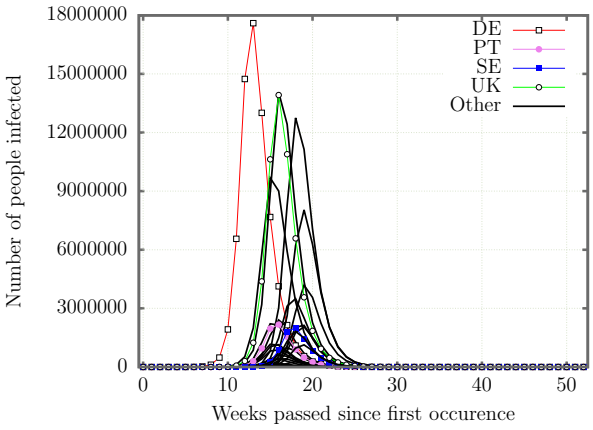

Figure 3: DE as initial disease host. As with SE, most countries follow closely behind as a cluster.

As both fig. 2 and fig. 3 signifies however, there can be no conclusive result drawn about the pattern of spread, since the high international exchange rate of people seems to cause an immediate cluster-reaction, independently of where the initial spread takes place. It must however be noted that the dynamics certainly can be affected slightly, as both SE and DE shift similarily

depending on who happens to take on the role of first target. Aside from that, other countries display notably similar behaviour between the two settings, and are, barring SE and DE, fairly indistinguishable.

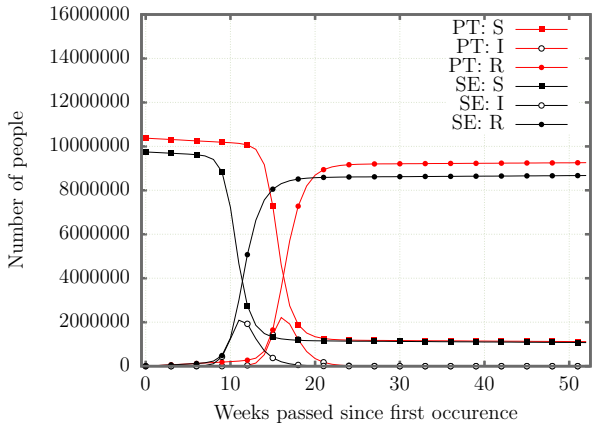

Figure 4: Disease dynamics for SE and PT, where SE is the country of first spread. Note the emphasised delay for the rapid increase in infected people between the countries.

One can in fig. 4 witness the aforementioned dynamics of an outbreak, where an outbreak in SE causes PT to exhibit its peak rougly six weeks later, whereas SE at that rate has nearly eradicated the disease. Furthermore, for both separate countries, it is clear that the dynamics are uniform in the sense that as soon as the disease has reached its peak, it is bound to soon be eradicated.

Interestingly enough, every country does in the end reach a state of equilibrium, where all derivatives in our model tend to zero. We can however not draw any conclusions between the quotas of the different parameters due to the travel terms; countries might by chance for example end up with large parts of their population pool labelled as a specific category, simply due to a large influx of sick individuals – an obvious limitation to our model with respect to reality.

Conclusions

Of all the countries investigated, there is indeed a bias depending on where the first outbreak takes place. There can, however, not be said that this causes any direct patterns that would imply any certain connections between countries in a more topological regard – all countries seem intertwined such

that a disease rather uniformly spreads out across the continent as soon as having left the country of first occurrence.

The effects are more easily measurable in the case of worst initial country, although the differences that our results display can not be said to be of very significant magnitude, since all values are of the same order. The difference could although be said to be relevant from a more humanitarian perspective, since it in absolute terms sums up to about 80000 people between the respective extremes.

Supposedly the lack of higher variance between countries is inherent to the choice of parameters, as perhaps it could be argued that a vaccination rate of 0.2 percent would be to high for a non-lethal disease, and that this would hamper the adequacy compared to an actual outcome.

The model does nevertheless behave rather realistically on the whole, as the apparent dynamics of our disease play in to produce a development that need not necessarily be too far from the truth.

Further research

Since the data accounted for is restricted to travel habits by flight, an extension to the model could certainly be made with regard to this. The SIR model does moreover not account for the dynamics that inherent to a population over a longer time span; for example deaths and births, which may be relevant to more severe outbreaks. It also does not account for the affectance of the disease on infected individual, who may change social habits and therefore not spread the disease uniformly throughout the stage of infection.

References

[1] Eurostat, *International intra-EU air passenger transport by reporting country and EU partner country*, http://ec.europa.eu/eurostat/data/database, retrieved 2015-09-09.

[2] Liao, C M, et al., *Probabilistic Transmission Dynamic Model to Assess Indoor Airborne Infection Risks*, Risk Analysis 25 (5): 1097–1107, 2005

[3] Tobias Ambjörnsson, *Lecture notes, Computational Physics*, Department of Astronomy and Theoretical Physics, Lund University, 2015.

Code

```
/* ODE solver utilizing a fourth-order Runge-Kutta method */
public final class ODEsolver
{
   private ODEsolver()
   {
   }
   // Calculates travel in between countries nodes n and m
   private static double travelSum(double[][] commute, double[]
      numppl, int n)
   {
      double sum = 0;
      for (int m = 0; m < numppl.length; m++)
      {
         sum += commute[m][n] * numppl[m] - commute[n][m] *
            numppl[n];
      }
      return sum;
   }

   /*
    * Takes in a network and a disease, utilizing these to compute
       a timeStep
    * through solving the to the model inherent ODE.
    */
   public static Network timeStep(Network network, Disease
      disease, double h)
   {
      double alpha = disease.getAlpha();
      double beta = disease.getBeta();
      double gamma = disease.getGamma();
      Node[] nodes = network.getNodes(); // nodes with initial
         values
      double[][] commute = network.getCommute();
      int nlen = nodes.length; // used a lot

      // Could much rather be a matrices ...
      double[][] kS = new double[4][nlen], kI = new
         double[4][nlen],
            kR = new double[4][nlen];
      double[] tempS = new double[nlen], tempI = new double[nlen],
            tempR = new double[nlen], tempN = new double[nlen];
```

```
double[] svals = new double[nlen], ivals = new double[nlen],
      rvals = new double[nlen];

// Saves away values for later use
for (int i = 0; i < nlen; i++)
{
   svals[i] = nodes[i].getS();
   ivals[i] = nodes[i].getI();
   rvals[i] = nodes[i].getR();
   tempS[i] = svals[i];
   tempI[i] = ivals[i];
   tempR[i] = rvals[i];
   tempN[i] = svals[i] + ivals[i] + rvals[i];
}

// First iteration
for (int i = 0; i < nlen; i++)
{
   kS[0][i] = h
         * (-tempS[i] * (beta * tempI[i] + gamma * tempN[i])
            / tempN[i]
               + travelSum(commute, tempS, i));
   kI[0][i] = h
         * (tempI[i] * (beta * tempS[i] - alpha * tempN[i]) /
            tempN[i]
               + travelSum(commute, tempI, i));
   kR[0][i] = h * (gamma * tempS[i] + alpha * tempI[i]
         + travelSum(commute, tempR, i));
}

// Calculates the new values of the temp arrays (for
   calculation of k2)
for (int i = 0; i < nlen; i++)
{
   tempS[i] = svals[i] + .5 * kS[0][i];
   tempI[i] = ivals[i] + .5 * kI[0][i];
   tempR[i] = rvals[i] + .5 * kR[0][i];
   tempN[i] = tempS[i] + tempI[i] + tempR[i];
}

// Second iteration
for (int i = 0; i < nlen; i++)
{
```

```
   kS[1][i] = h
         * (-tempS[i] * (beta * tempI[i] + gamma * tempN[i])
           / tempN[i]
              + travelSum(commute, tempS, i));
   kI[1][i] = h
         * (tempI[i] * (beta * tempS[i] - alpha * tempN[i]) /
           tempN[i]
              + travelSum(commute, tempI, i));
   kR[1][i] = h * (gamma * tempS[i] + alpha * tempI[i]
         + travelSum(commute, tempR, i));
}

for (int i = 0; i < nlen; i++)
{
   tempS[i] = svals[i] + .5 * kS[1][i];
   tempI[i] = ivals[i] + .5 * kI[1][i];
   tempR[i] = rvals[i] + .5 * kR[1][i];
   tempN[i] = tempS[i] + tempI[i] + tempR[i];
}

// Third iteration
for (int i = 0; i < nlen; i++)
{
   kS[2][i] = h
         * (-tempS[i] * (beta * tempI[i] + gamma * tempN[i])
           / tempN[i]
              + travelSum(commute, tempS, i));
   kI[2][i] = h
         * (tempI[i] * (beta * tempS[i] - alpha * tempN[i]) /
           tempN[i]
              + travelSum(commute, tempI, i));
   kR[2][i] = h * (gamma * tempS[i] + alpha * tempI[i]
         + travelSum(commute, tempR, i));
}

for (int i = 0; i < nlen; i++)
{
   tempS[i] = svals[i] + kS[2][i];
   tempI[i] = ivals[i] + kI[2][i];
   tempR[i] = rvals[i] + kR[2][i];
   tempN[i] = tempS[i] + tempI[i] + tempR[i];
}
```

```
      // Fourth iteration
      for (int i = 0; i < nlen; i++)
      {
         kS[3][i] = h
               * (-tempS[i] * (beta * tempI[i] + gamma * tempN[i])
                  / tempN[i]
                     + travelSum(commute, tempS, i));
         kI[3][i] = h
               * (tempI[i] * (beta * tempS[i] - alpha * tempN[i]) /
                  tempN[i]
                     + travelSum(commute, tempI, i));
         kR[3][i] = h * (gamma * tempS[i] + alpha * tempI[i]
               + travelSum(commute, tempR, i));
      }

      // Calculates the values for the next timestep
      for (int i = 0; i < nlen; i++)
      {
         nodes[i].setS(svals[i] + kS[0][i] / 6 + kS[1][i] / 3 +
           kS[2][i] / 3
              + kS[3][i] / 6);
         nodes[i].setI(ivals[i] + kI[0][i] / 6 + kI[1][i] / 3 +
           kI[2][i] / 3
              + kI[3][i] / 6);
         nodes[i].setR(rvals[i] + kR[0][i] / 6 + kR[1][i] / 3 +
           kR[2][i] / 3
              + kR[3][i] / 6);
      }
      network.setNodes(nodes);
      return network;
   }
}
```

```
import java.io.*;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;

/* Main method simulating the spreading of a disease over a
   network
 * consisting of a set of nodes. */
public class DiseaseSpread
{
```

```
private final static double h = 0.01; // size of the timestep

public static void main(String[] args) throws
   FileNotFoundException
{
   Network europe = new Network();
   ArrayList<Node> nodes = new ArrayList<Node>();
   Disease disease = new Disease(1, 2.5, .002);
   BufferedReader br = new BufferedReader(new
      FileReader(args[0]));
   double[][] commute = new double[28][28]; // travel weights
   int iterations = 52;

   // Read from file and compute
   String line = null;
   try
   {
      line = br.readLine();
   } catch (IOException e)
   {
      e.printStackTrace();
   }
   Set<String> seen = new HashSet<String>();
   while (line != null)
   {
      String[] fields = line.split("\t");
      if (!seen.contains(fields[0]))
      {
         seen.add(fields[0]);
         nodes.add(new Node(fields[0],
            Double.parseDouble(fields[3]), 0, 0));
      }
      for (int i = 0; i < 28; i++)
      {
         if (nodes.size() - 1 == i)
            continue;

         String[] innerFields = line.split("\t");
         commute[nodes.size() - 1][i] =
            Double.parseDouble(innerFields[2])
               / Double.parseDouble(innerFields[3]) / 52.;
         try
         {
```

```
            line = br.readLine();
         } catch (IOException e)
         {
            e.printStackTrace();
         }
      }
   }
   try
   {
      br.close();
   } catch (IOException e)
   {
      e.printStackTrace();
   }

   europe.setCommute(commute);
   nodes.get(0).setI(1); // Set initial infected
   europe.setNodes(nodes);
   Node[] nodearr = europe.getNodes(); // Used a lot
   PrintWriter[] w = new PrintWriter[europe.getNodes().length];

   System.out.println("#Week \tSusceptible \tInfected
      \tRecovered");
   for (int i = 0; i < nodearr.length; i++)
   {
      w[i] = new PrintWriter(
            new FileOutputStream(nodearr[i].getName() + ".dat"),
               true);
      w[i].println("#t \tSusceptible \tInfected \tRecovered");
      w[i].println("0 \t" + (int) (nodearr[i].getS() + .5) +
         "\t"
            + (int) (nodearr[i].getI() + .5) + "\t"
            + (int) (nodearr[i].getR() + .5));

      System.out.println("0 \t" + (int) (nodearr[i].getS() +
         .5) + "\t"
            + (int) (nodearr[i].getI() + .5) + "\t"
            + (int) (nodearr[i].getR() + .5));
   }

   for (int i = 0; i < iterations / h; i++)
   {
      Network newNetwork;
```

```
            newNetwork = ODEsolver.timeStep(europe, disease, h);
            europe.setNodes(newNetwork.getNodes());

            // Saves values to files and prints to console
            for (int j = 0; j < newNetwork.getNodes().length; j++)
            {
            w[j].println(((i + 1) * h) + "\t" + (int)
               (nodearr[j].getS() + .5)
                  + "\t" + (int) (nodearr[j].getI() + .5) + "\t"
                  + (int) (nodearr[j].getR() + .5));
            System.out
                  .println((i + 1) * h + "\t" + (int)
                    (nodearr[j].getS() + .5)
                       + "\t" + (int) (nodearr[j].getI() + .5) + "\t"
                       + (int) (nodearr[j].getR() + .5));
            }
         }
         for (int i = 0; i < w.length; i++)
            w[i].close();
      }
   }
```

```
public class Disease
{
   private double alpha; // recovery rate
   private double beta; // infection rate
   private double gamma; // vaccination rate

   public Disease(double a, double b, double c)
   {
      this.alpha = a;
      this.beta = b;
      this.gamma = c;
   }

   public double getAlpha()
   {
      return alpha;
   }

   public double getBeta()
   {
      return beta;
```

```
   }

   public double getGamma()
   {
      return gamma;
   }

}
```

```
import java.util.ArrayList;

/* Takes care of all nodes. */
public class Network
{
   private ArrayList<Node> nodes;
   private double[][] commute;

   public Network()
   {
   }

   public Node[] getNodes()
   {
      Node[] nodearr = new Node[nodes.size()];

      for (int i = 0; i < nodes.size(); i++)
         nodearr[i] = nodes.get(i);
      return nodearr;
   }

   public void setNodes(Node[] nodearr)
   {
      ArrayList<Node> newNodes = new ArrayList<Node>();
      for (Node node : nodearr)
         newNodes.add(node);
      nodes = newNodes;
   }

   public void setNodes(ArrayList<Node> nodes)
   {
      this.nodes = nodes;
   }
```

```
   public void addNode(Node node)
   {
      nodes.add(node);
   }

   public double[][] getCommute()
   {
      return commute;
   }

   public void setCommute(double[][] commute)
   {
      this.commute = commute;
   }

}
```

```
/* Class representing a node in a network. */
public class Node
{
   public String name;
   private double S, I, R;

   public Node(String name, double S, double I, double R)
   {
      this.name = name;
      this.S = S;
      this.I = I;
      this.R = R;
   }

   public double getS()
   {
      return S;
   }

   public double getI()
   {
      return I;
   }

   public double getR()
   {
```

```
        return R;
    }

    public double getN()
    {
        return S + I + R;
    }

    public String getName()
    {
        return name;
    }

    public void setS(double S)
    {
        this.S = S;
    }

    public void setI(double I)
    {
        this.I = I;
    }

    public void setR(double R)
    {
        this.R = R;
    }

}
```