**Intro-y abstract**

Agent-based models play a useful role in the social sciences, helping us to understand emergent patterns in social and environmental systems while retaining detail of the individuals. However, their predictive potential is limited, as calibration is often limited to one-shot calibration. The purpose of this paper is to act as a simple how-to guide for applying a method of dynamic data assimilation (the ensemble Kalman filter) to a basic agent-based model. A simple example is explored, and the results show how this method can be used to achieve efficient successful sequential parameter simulation.

**Background about ABMs, why we need DA, what methods are already being done**

Agent based models are models where the modeller: identifies active agents, defines their behaviour, puts them in a certain environment, defines interactions (both between multiple agents and between agents and the environment), and runs the simulation to see what happens. They’re particularly useful for studying emergent behaviour of systems composed of interacting agents, while still retaining information about the individuals. Agent-based models are used in a variety of fields for a number of different purposes, but are perhaps most well-suited to describing social and environmental systems.

Currently, however, agent-based models are often calibrated only once using a past set of training data, and often the results are not validated against future observations [ref][ref], restricting their predictive potential. Common methods for calibrating agent-based model include sampling the model parameters using simulated annealing and genetic algorithms, and then seeing how the data generated using these parameters compares with the real data. A lot of information and guidance exists on how to apply these forms of calibration [ref],[ref],[ref], but no literature exists yet which details in a clear and simple way how one might apply alternative, real-time forms of calibration: namely data assimilation.

Data assimilation is the process of combining observed data of a system with forecasts generated by a numerical model of that system. The model state is typically updated to account for the observations, and then this updated state is used as the basis to generate more forecasts. The process repeats; the model is calibrated on the fly, resulting in timely and continuous error constraint. [ref]

Data assimilation of agent-based models is an as-yet unexplored area. Typically the agent-based modelling community has been limited by the availability of data, but the recent emergence of streamed ‘big’ datasets means that data assimilation could now have transformative effects on the predictive power of agent-based models. For this reason we have applied one data assimilation method, the ensemble Kalman filter, to a very basic agent-based model; we did this with the intention of exploring ABM-specific problems with data assimilation. Our aims with this paper are to introduce the ABM community to the concepts of data assimilation and the ensemble Kalman filter process, and to outline how to apply these kind of methods to a basic model. Hopefully individuals can then extend these concepts to more complex models appropriate to their own research.

**The (ensemble) Kalman filter**

The ensemble Kalman filter is a standard method of data assimilation. The algorithm combines a series of noisy observations made over time with a model that has knowledge of the underlying system, to produce estimates of the true state of the system. The noisy observations are combined with the model forecasts, and their influences on the adjusted estimates are weighted relative to the uncertainty of each. The adjusted estimates produced should ideally be more accurate than the noisy observations.

The ensemble Kalman filter involves an ensemble of estimated model states. The information about these model states are held in state vectors. This ensemble of state vectors is initialised either with knowledge, or estimates, of the original system state and its uncertainty.

Each model state is then evolved forward independently according to two steps:

1. The forecast step:  
   The estimated model state is fed into the model, and the model plays forward until the next observation time, generating a forecast. The ensemble forecast states mean will give an estimated forecast of the true state, while the covariance of the ensemble forecast states provides a measure of its uncertainty.
2. The data assimilation step:  
   Upon receiving the actual observation, the ensemble forecasts are updated accordingly. The updated values are called the ensemble analysis. As before, the ensemble analysis mean will give the best estimation of the true state, while the covariance of the ensemble analysis states provides a measure of its uncertainty.

These steps can repeat each time a new observation is made.

By including parameters in the state vector, these are also updated during the iterative process through their covariance with the observables. Predicting these unknown parameters this way is known as sequential parameter estimation.

**My model**

The concept for the model stemmed from my previous work: I had previously investigated publicly available footfall data sourced from cameras in Leeds city centre, to answer the question of whether it’s possible to deduce the origin-destination matrix of the people in the system. An agent-based model was designed in python to model agents moving to and from work each day across a map of Leeds city centre, with cameras counting the agents as they pass. A set of camera footfall counts were generated using a test origin-destination matrix, and a genetic algorithm was used to find the origin-destination matrix given the generated camera counts alone. Multiple, incredibly varied solutions were found, under several different camera configurations, suggesting that the resolution of the data set (counts recorded every hour) is not fine enough to restrict the origin-destination matrix for each set of camera counts.

For the sake of effectively applying and exploring data assimilation methods in a basic case, this model was greatly simplified. The map was reduced to one entrance point (point A, coordinates 0,1) and one exit point (point B, coordinates 0,40), separated by a 1D line. Active agents move from point A to point B, moving one coordinate along each iteration of the model.

When an agent is not active, it is located at a point C (coordinates 0, 0). The reason for keeping these agents in the system, rather than simply deleting them from the program’s memory, will become apparent when applying the Kalman filter later.

Each iteration of the simulation corresponds to a minute of time. Every 60 iterations (minutes) a certain number of agents are activated and moved from point C to point A. Every iteration these agents then move to the next coordinate. At a point D (arbitrarily chosen with coordinates 0, 24), some of the agents will instantly return to point C at a certain rate termed the bleed out rate. Those agents that pass through point D without returning instantly to point C will continue their journey to point B. Once they’ve reached point B, they will return to point C until they are reactivated at a later time.

Footfall cameras are also included in the model; positioned at points A and B, they record the number of agents leaving these positions and return hourly footfall counts.

The number of agents being activated at point A is set for each hour of the day, and does not vary day to day. The distribution throughout the day goes roughly as a normal distribution, peaking at midday. Figure 1 shows an example run of 5 days. The blue counts are camera A counts at point A, and the green counts are camera B counts at point B.

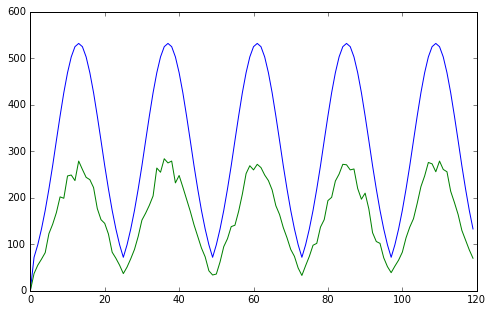


Fig.1. Graph showing typical camera counts for cameras A and B. The blue line shows camera A counts, and the green line shows camera B counts. These counts were generated by a bleed out rate of 0.5: as expected, the camera B counts are roughly half that of the camera A counts, and show random variation from the strict normal distribution of the camera A counts.

The bleed out rate in this simple case does not change at all throughout the simulation.

We will assume (unrealistic) knowledge of the distribution of agents being activated at point A, and attempt to use data assimilation to estimate the true number of agents at point B – thereby performing parameter estimation of the bleed out rate.

**How I applied the Kalman filter**

As this is such a simple ABM and we had no real world analogue to hand, synthetic data was generated. The bleed out rate was drawn from a normal distribution of 0.5, standard deviation 0.1, and the model was played forward 7200 minutes (5 days). This ‘truth’ data was stored.

The state vector has to contain all information necessary to step the model forward. In this case this meant it had to contain: the location of each agent, how far along its journey it was, the bleed out rate, and the camera counts at camera A and camera B. With a system containing 600 agents, this works out to be a 1203x1 column vector.

Since the state vector remains the same size during the ensemble Kalman filter process, location information about all the agents has to exist at every iteration. For this reason we need to keep all the agents located in the system somehow, even if they’re not actively engaging with the system. Luckily for this basic system we are assuming knowledge of the camera A counts; therefore we know the maximum number of agents that will be active at any one time (no more than 600) and can set this as our total number of agents. However, in a system that didn’t have this information, such as one where the number of agents at camera A was random, this proves a much larger problem.

The observation vector consists of the observables in the system: the camera counts at camera A and camera B. These counts were generated by drawing the values from the truth data, and then adding normal random noise to create ‘virtual observations’. The observation vector in this case is a 2x1 column vector. The random noise errors were drawn from a normal distribution with mean 0, and an initial standard deviation of 5 (this was adjusted to 10 and then 15 in later runs of the Kalman filter).

The ensemble was initialised by drawing 30 bleed out rates, from the same normal distribution that the true parameter was chosen from: mean 0.5, standard deviation 0.1. The model was then run forward an hour for each of these rates. The resulting 30 state vectors (the forecasts) were stored. The ensemble means were calculated, and the variance of each value informed the covariance matrix P. We now have our forecast means (including the forecast camera B counts), and our forecast variance.

Next the data assimilation step begins. The observation vector is extracted from the truth data, and the virtual observations are generated. The matrix R which represents the uncertainty in the observation varied from diag(5^2,5^2) to diag(15^2, 15^2) to match the normally distributed noise.

The matrix H is just a transformation matrix that changes the state vector into the same form as the observation vector. In this example, with a 2x1 observation vector and a 1203x1 state vector, it is a 2x1203 matrix with 1s in the correct positions to pick the camera counts out of the state vector.

Next we calculate the Kalman gain matrix. This is some straightforward matrix manipulation in numpy:

From here we can now create the ensemble analysis. Using the formula:

Analysis = forecasts + K.(virtual obs – H.forecasts)

We can now take the mean to find the analysis mean, and using this work out the analysis variance. As before, this informs the analysis covariance matrix.

The analysis state vectors are then fed into the model, forecasted forward an hour, and the resulting state vectors provide the new forecasts. The next item of data is measured (in our case extracted from the ‘truth’ data) and the process repeats.

The system has also neatly avoided another potential problem. As the agents are only activated at point A every hour, and the journey from point A to point B takes less than an hour, all the agents will have returned to point C before the next lot of agents are activated at point A. This means that across the ensemble all the agents will have the same locations: either they’ve just been activated at point A (0,1) or they’re at point C (0,0). This leads to an ensemble analysis mean that is always an integer, and is a possible configuration of the ABM. However, this may not hold for more complex ABMs. This also highlights that the Kalman filter is not an appropriate method of data assimilation if there are discontinuous variables in the system; for example, if the agents were capable of randomly transporting from coordinate (0,5) to (0,50) and the space between was not a valid model configuration. The ensemble Kalman filter method may result in forecast and analysis means that make no sense in the context of the model. Unfortunately ABMs often include discrete variables, limiting the ensemble Kalman filter’s usefulness.

**The results, was it actually any good?? Does it get closer to the truth than the virtual obs do?**

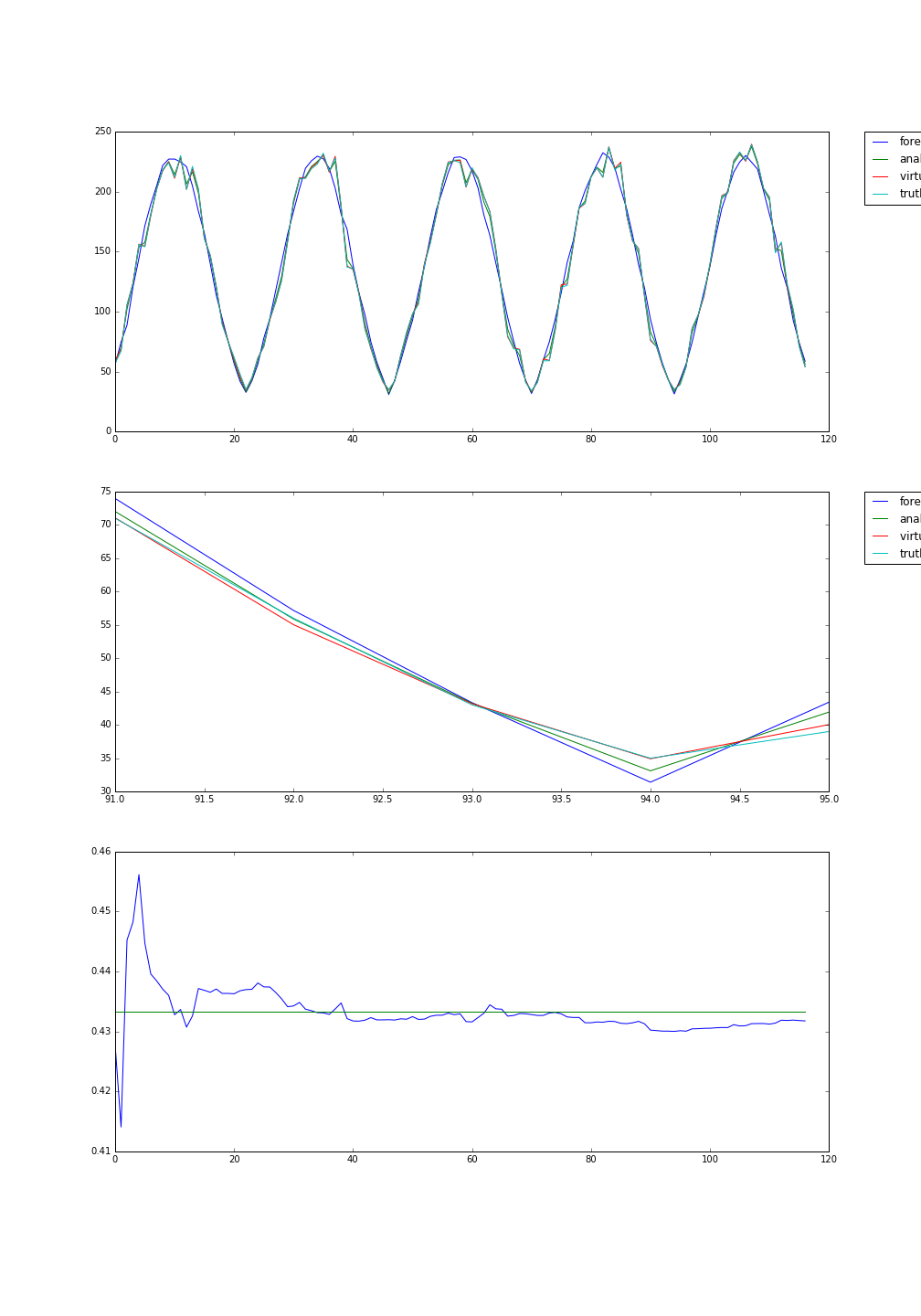
In figure 2, we illustrate the results obtained when using a virtual observation error of mean 0, standard deviation 5. In 2.a) only the forecast and the analysis are clearly distinguishable – the analysis appears so close to the truth and virtual observations that this level of resolution doesn’t show a clear difference. 2.c) shows the bleed out rate changing with time. The fact that the Kalman filter quickly hones in on an accurate value for the bleed out rate is clearly reflected in the overall forecasts, which in2.a) can be seen to mirror the ground truth well; the deviations from the normal distribution that occur due to stochasticity are naturally not picked up on by the ensemble, which will always ultimately average to an approximate normal distribution. 2.b) shows a zoomed in view of a period when the parameter estimation is close to the truth. For a period of time here the analysis is closer to the ground truth than the virtual observations.

However, looking at the RMSE for the forecast, analysis and virtual observations, we find a forecast RMSE: 9.80, analysis RMSE: 2.58 and observation RMSE: 0.91. We would expect the analysis RMSE to be less than the observation RMSE, as knowledge of the underlying system should improve noisy observations.

One reason for this could simply be the randomness of the model. Knowledge of the underlying system helps to guide knowledge of camera B, but ultimately the data relies so heavily on randomness that these intelligent guesses will rarely be closer than observations (even noisy observations).

This aside, the parameter estimation does well. After a period of initial fluctuation the bleed out rate settles to a roughly constant value, varying slowly from ~0.427 – ~0.435. Given that the true bleed out rate is 0.433, this is fairly successful.

Figure 3 shows the parameter estimation results for three runs, with virtual observation standard deviations 5, 10 and 15 respectively. All three do reasonably well, with no discernible trend in accuracy due to observation error.

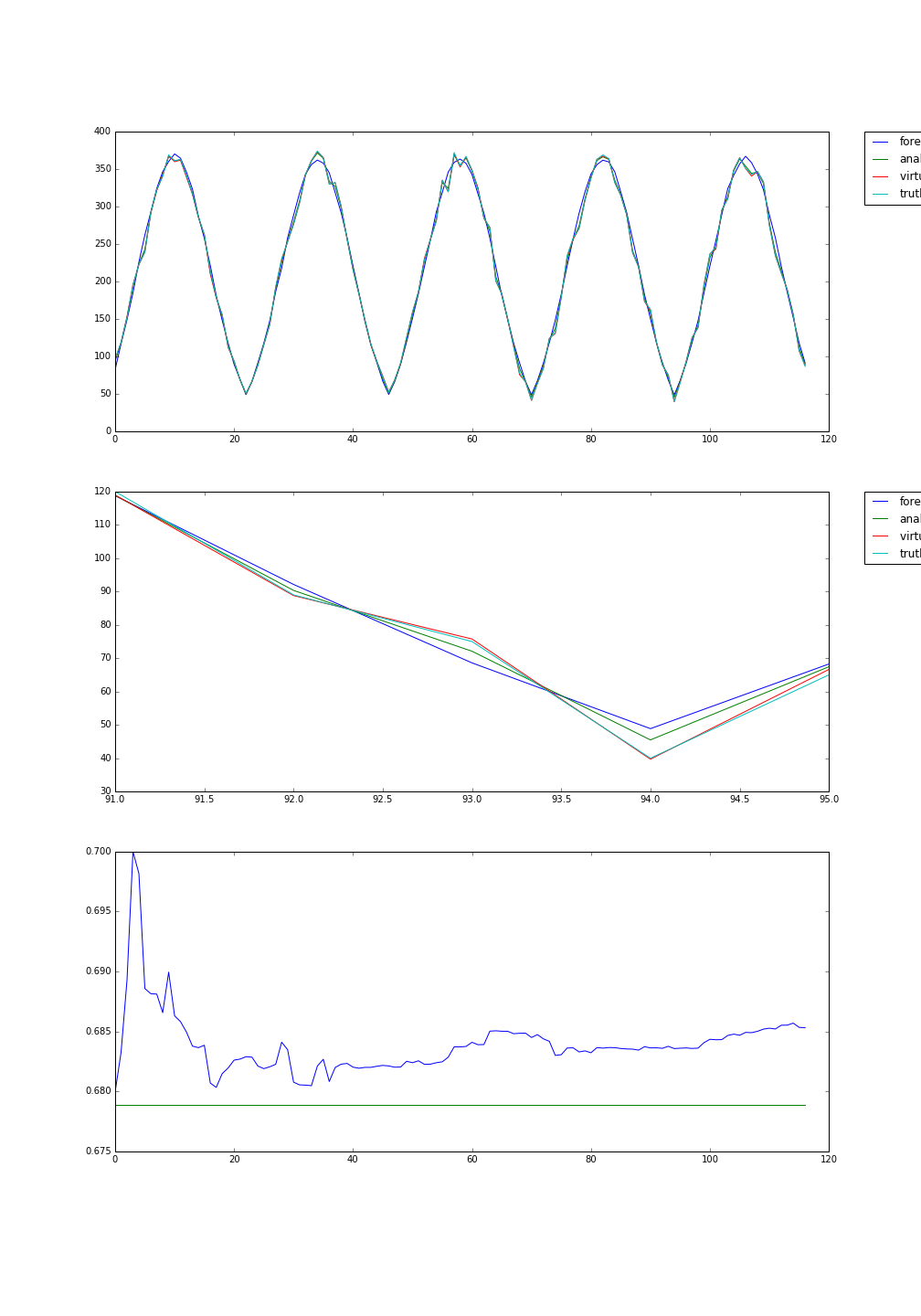


2.a)

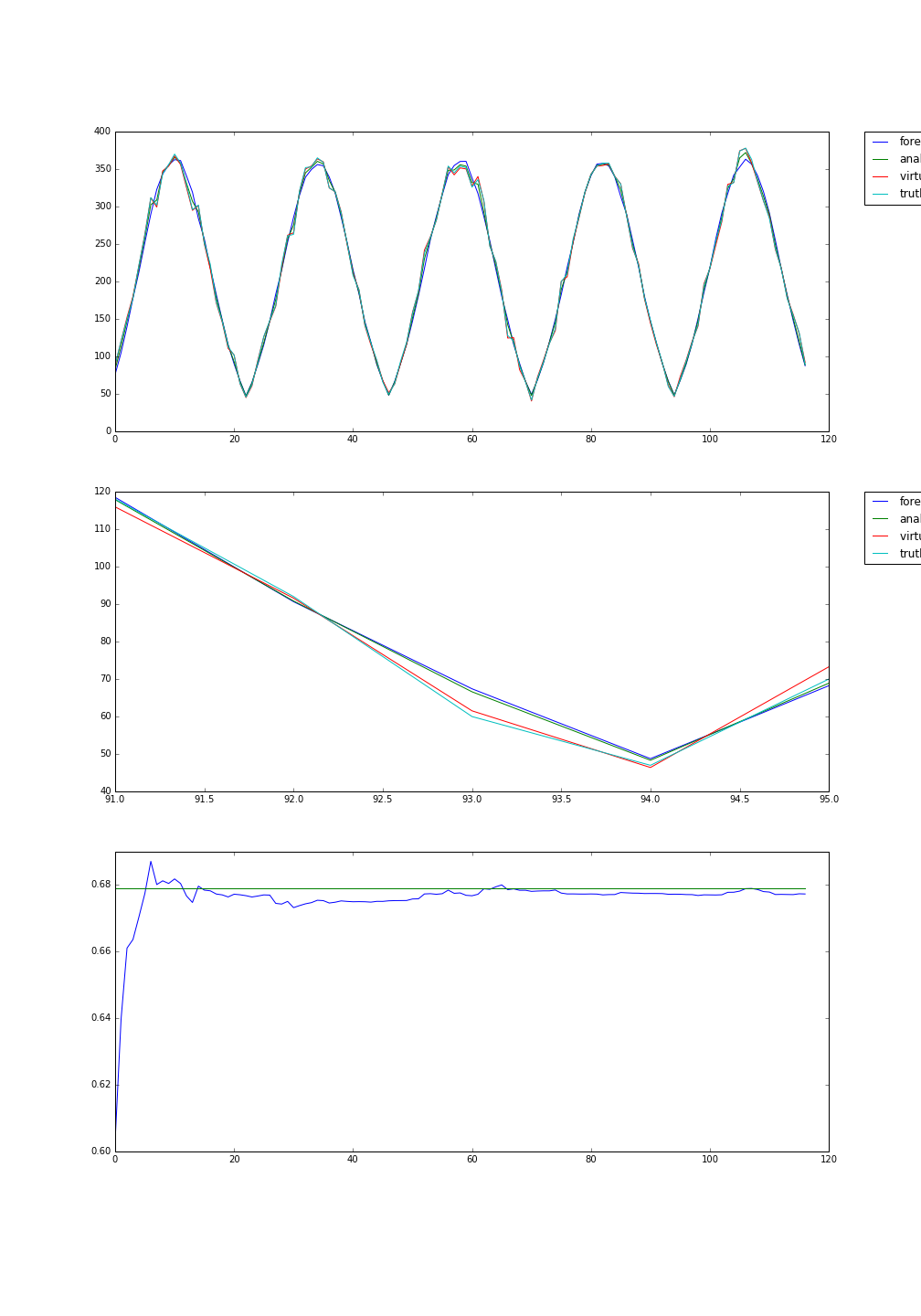
2.b)

2.c)

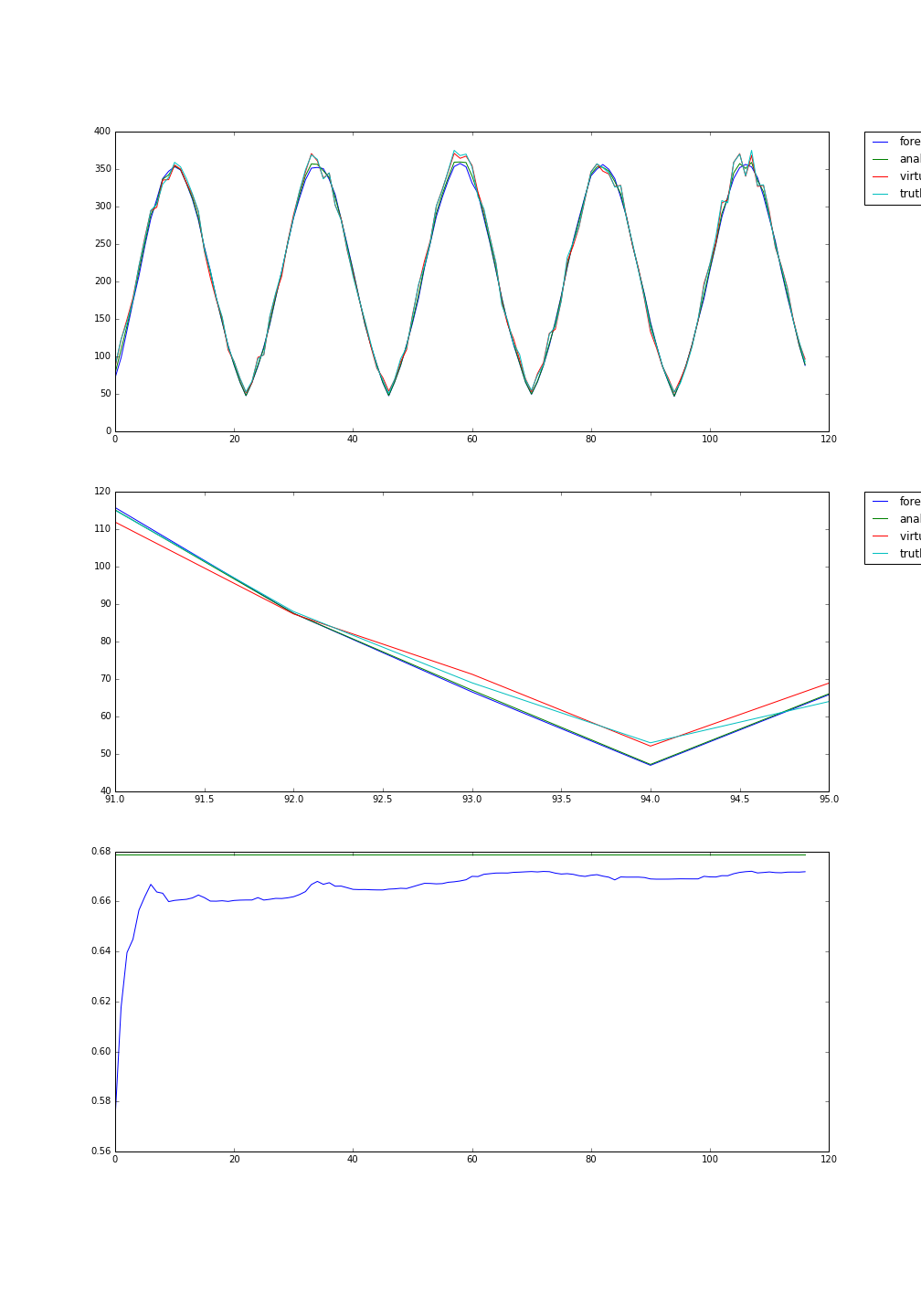
Fig.2. Graphs showing the Kalman filter at work for 5 days. 2.a) shows the forecast, analysis, virtual observations and ground truth all on one time series plot. 2.b) shows is a zoomed in look at part of 2.a), making clearer the distinction between the four different time series. 2.c) shows the results of the sequential parameter estimation: the green line shows the value of the true bleed out rate, used to generate the ground truth, while the blue line shows the value of the bleed out rate in the state vector.

****

3.a)

****

3.b)

****

3.c)

Fig.3. Graphs showing the sequential parameter estimation for virtual observations with error mean 0, standard deviation 5 (3.a), 10 (3.b) and 15 (3.c). If anything the parameter estimation appears to be the least accurate for the instance where the standard deviation is smallest.