EOLDAS Users' Documentation

Release 2012.06

P Lewis, J Gomez-Dans

July 18, 2012

CONTENTS

Note: A PDF version of this document is available here

CHAPTER

ONE

INTRODUCTION

The EOLDAS project is funded by ESA as part of the Support to Science Element (STSE) of the Earth observation Envelope Programme.

The aims of the projects are:

- To develop and document a generic data assimilation scheme to assist the retrieval of geophysical parameters from medium resolution optical EO data
- To develop a prototype software package implementing selected aspects of the scheme
- To validate the prototype using multi-sensor EO data and field measurements

The project is led by the UK National Centre for Earth Observation (NCEO) with a team that includes:

- University College London
- The University of Reading
- The European Commision's Joint Research Center
- Friederich Schiller Universität Jena
- FastOpt GMbH
- Assimila

CHAPTER

TWO

CONTENTS

2.1 Introduction

This webiste contains a **live** version of the EOLDAS documentation. EOLDAS stands for Earth Observation Land Data Assimilation System, and in here we refer to an implementation of a Data Assimilation (DA) prototype to be used to monitor the land surface in the optical domain. The code is fairly generic, and can be used to implement a number of different variational DA schemes (although we mostly have used weak constraints)

This software was developed

Note: Partners, project number, thanks to etc.

2.1.1 Installation

Requirements

EOLDAS has mostly been written in Python, with some radiative transfer codes written in Fortran and made available to the main Python library. The requirements for the package are

- Python (versions > 2.5 and < 3.0)
- Scipy
- Numpy
- Matplotlib
- gfortran
- OpenOpt (optional)
- git (version control system. Optional)

The first few packages are usually available in modern Linux distributions, as well as in MacOsX. In some cases, the use of the Enthought Python Distribution to install a whole Python ecosystem might be a worthy option. Similar efforts for Windows might be found in Python xy.

OpenOpt is an optimisation suite, and is optional, although its use is highly recommended. Using `pip` or `easy_install`, the Python package installers, it can be installed as

```
easy_install -U openopt
```

or

```
pip install openopt
```

Note that if you do not have root/superuser access, you can install it for your user as

```
easy_install --user -U openopt
```

or

```
pip install --user openopt
```

Installation

The eoldas distribution can be installed using the same approach as OpenOpt:

```
easy_install --user -U eoldas
```

or:

```
pip install --user eoldas
```

This will install as an user, ignore the `--user` option to do a system-wide install. Note that executables will be installed too, and the installation executables path will be installation dependent. Note that this command will also install the semidiscrete radiative transfer code and compile it (many warning will show up, but they are safe to ignore).

Installing from source

If you plan to do development on EOLDAS, we strongly encourage you to fork the EOLDAS repository on github, and feel free to submit patches, suggestions, etc. You can also download a snapshot of the code from github. Once you have unpacked the tarball, you can install it using the following command

```
python setup.py install
```

(as before, if you don't have super user priviledges, add the `--user` flag to the previous command).

Installing the user's guide

The user's guide is available on github.com. Although you can just download an archive with the examples, data and source of this guide, we encourage you to fork the eoldas_release git repository, and to work on it and submit bug reports, fixes etc.

Getting help & Collaborating on improving EOLDAS

EOLDAS is a complex and varied tool that can help solve many problems where Gaussian statistics are a good assumption in variational problems. We would like users to use github as a way to collaborate in fixing bugs, spreading what you can do with EOLDAS, and to put together tutorials.

There are two main ways for collaboration in either the user's guide or the main EOLDAS package: one is to make a pull request after you have forked the relevant repository (`eoldas` or `eoldas_release`). A second avenue is to use the Wiki. The user's guide has Wiki page, and so does the main eoldas repostiory.

2.2 A tutorial guide of the EOLDAS prototype software

2.2.1 Purpose

This document provides a guide to concepts of Data Assimilation (DA) and the use of the EOLDAS prototype software.

EOLDAS is written in Python and provides an easy way to implement and test out advanced DA concepts. It should be portable to most computer systems with a sane Python installation and a FORTRAN compiler.

The primary purpose of the EOLDAS software is to allow the optimal estimation of (mainly land surface, mainly vegetation) state variables (e.g. quantifying how much vegetation there is at a particular time and place) from remote radiometric (Earth Observation) data.

A secondary task, that will be further explored in a follow on project and within NCEO is to allow us to use Earth Observation data to test and constrain physical models of how we think our environment 'works', encoded as what me can call 'process models'.

This is not an easy task, as there is generally not enough 'information' in our observations alone to enable a 'good estimate' of vegetation quantities from space-based observations. In any case, whatever approach we take, we want to be able to have an idea of how 'good' our estimate is.

We find that Data Assimilation, an optimal estimation framework allows us to apply multiple constraints in a consistent manner to achieve our aim.

2.2.2 Introduction to the problem we are trying to solve with an EOLDAS

In recent years, monitoring of the Land surface has received great attention. Accurate estimates of the state of the vegetation, snow cover or disturbance are required by scientists and decision-makers in fields as diverse as environmental science, disaster monitoring or food security. Earth Observation (EO) data, being (mostly) radiometric measurements from instruments on satellite platforms, have great potential to to contribute to such monitoring. Further, we have historical archives of EO data over several decades that allow us to start to build a picture of how physical properties on the Earth surface have been changing. We can then use these data for instance to test the accuracy and robustness of 'models' that are implementations of our current understanding of how people and the Earth System interact. We can then contribute to improving these modelsand improving our understanding.

The main advantage that EO has is that it is ia consistent source of information (provided instrument calibration is understood and monitored) and can often provide global coverage. In a sense, it is 'objective' (within the limits of the orbital and physical measurement constraints imposed on it), and it is also (certainly for the user at leasti, and also arguably because of the coverage it provides) generally quite a cheap source of information. Probably one of the most common use of land surface EO data that the general public come into contact with is that available through sources such as Google Earth. Such data provide 'visual' information to the user, and with a little 'photo interpretation', many properties of the land surafce can be inferred.

The problem we have here though is to 'automatically' provide reliable and constent information from such data for global coverage: we want to quantify the properties of the land surface, not just to describe it. The EO data that we have access to do not directly provide the information we want. If we want to improve models of e.g. how vegetation interacts with climate, we need to connect the radiometric information to what we can call model 'state variables': things that the models have a representation of, such as the amount of leaf area per unit land area. This implies some sort of inference.

Traditionally in land surface EO and many other areas, we have taken the apporoach of trying to estimate these state variables directly from the observations, and to provide 'products', such as those developed and distributed by national Space Agencies and other organisations to users. This is a laudable and valid aim and has been practically achievable since the start of this millenium. It has allowed 'end users' access to vast amount of information that have made significant contributions to monitoring and understanding how the Earth System 'works' and what the impacts of people and climate (as major influencing factors) have been.

These products have, in the main, been developed to provide consistent datasets from a single source of information. So, for example, a global leaf area index (LAI, the amount of one sided leaf area per unit ground area) dataset is generated by NASA from the MODIS intrument and others from projects such as Geoland2 from the EU. There are many practical reasons for developing such products using data from individual satellite sensors, but each product has its own set of issues which the developers try to quantify and encapsulate in quality assurace (QA) information. This is generally available for each spatial and temporal sample in the dataset, but at the moment, it is more difficult to know what the uncertainty of any particular observation might be. There has been much work on 'validating' and (inter-)comparing these products to try to gauge how well they agree, but it it problematic to combine them without loss of information.

Further, and potentially significant for many end users who really just want to know the LAI over some area and time (and how good that estimate is), each product is generated using different assumptions about the surface they

suppose to measure. If these data are to be ingested into other models that have an LAI state, close inspection of the models used to generate the EO datasets and those in the process models will often reveal a mis-match in the assumtpions made (e.g. concerning the spatial distribution of vegetation elements: whether and how ther are clumped or not). Additionally, quantifying uncertainties in the data is far from trivial as it involves considering the impact of the whole collection of information used: the measurements, the models used for inference etc.

This leads us on to a Bayesian idea of the 'data' (information) we are trying to produce. In essence, we cannot provide a 'truth', but rather an estimate of the likely statistical distribution of the information we infer. Within such an approach, we can combine information using Bayes rule. This allow us to (potentially) explicitly consider and combine the various sources of uncertainty in our mapping from EO data to information. This can include information beyond that in the EO data themselves.

It is now established that an optimal blending of these different sources of information leads to the best possible estimate of the state of the land surface. This estimate needs to be qualified, and provide information on how uncertain it is, or how different parts of the system co-vary.

Data assimilation (DA) techniques allow the combination of different sources of information, weighted by their individual uncertainty, to provide an estimate of the state of the land surface that is conditional on all the information that is combined. The EOLDAS project aims to demonstrate DA techniques with satellite data by developing a software tool that allows the exploration of DA techniques. This requires the usage of complex physical models that explain how the scene of interest is translated into the actual radiometric measurements, as well as ancillary information of how some parameters may vary, either through mechanistic models (for example, dynamic vegetation models that explain the development of vegetation using simple rules based on typically meteorological inputs), statistical models (for example, historical estimates of leaf area index from field measurements) or simple constraints of the evolution of these parameters (for example, by forcing the evolution of leaf area index to be smooth and continuous).

2.2.3 Capabilities and Limitations of the current EOLDAS

The EOLDAS software allows for various forms of EO and other data to be combined to provide an optimal estimate of state variables. It requires that the uncertainties of the input data are quantified. At present, this is limited to an assumption that Gaussian statistics can be used to describe the uncertainty, i.e. input data are represented by their mean value and standard deviation (or more fully a covariance matrix, although some of the input data formats currently accepted only consider standard deviation). This can cause problems for highly non-linear systems, especially if the uncertainties are large, so this can be partially overcome by transforming the representation of the state variable. Arbirtary transformations are allowed, but a typical one would be to work with a state variable that is an exponential transformation of the desired state variable.

The DA system requires a model to map from EO data (radiance or reflectance data) to the state variables of interest (e.g. LAI). We will call such models 'Observation Operators' as they map state to observations. A few example Observation Operators are available in this release of EOLDAS, including the semi-discrete radiative transfer model of Gobron et al. 1997, for which an adjoint has been develoepd here to allow more rapid state estimation. In addition, the linear Kernel models used in the MODIS BRDF/Albedo product are included and an interface to the 6S atmospheric model is provided. The EO data that can be used is then essentially limited to observations in the optical domain (~450-2500 nm), if these operators are used.

No 'biophysical' process models are included in this release, only a form of regularisation model.

2.3 Initial exploration

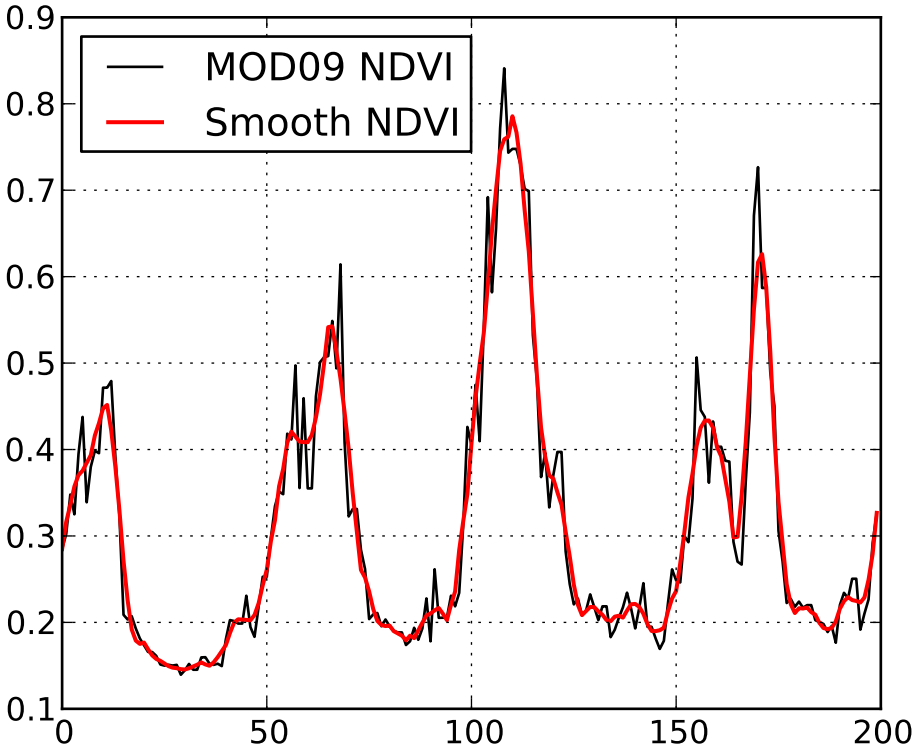
2.3.1 Exploring a univariate timeseries

EO data timeseries allow for the monitoring of the land surfaces. The data gathered by space-borne sensors can be related to a number of parameters of interest, such as "vegetation greenness", fraction of absorbed photosynthetically active radiation absorbed by a canopy, etc. Monitoring how a location on the Earth's surface changes with respect to time is thus an important task of remote sensing. Analysing these data is fraught with difficulties: as any other measurement technique, data have uncertainties. Cloud cover or the choice of orbit can result in data only being available for a few dates. The role of data assimilation (DA) is to use these measurements, together

with any other ancillary information that may be available on the state of the land surface, to infer its true state. This inference exercise should provide us with an idea of how much should we trust our estimate. This document presents examples of DA techniques, using the EOLDAS prototype software as a demonstration.

For the first example, we'll consider a time series of the normalised difference vegetation index (NDVI) over an agricultural area in southern Spain. NDVI has been correlated with vegetation development: the index is low when the scene is composed mostly of bare soil and rises as the crop develops. The VI drops again as the crop senesces and is eventually harvested. It is interesting to us as an univariate time series with quite a clear shape. We can see four years of data, with a fallow/vegetative period cadence. We note, however, that the series is noisy, with abrupt changes and spikes showing up. This variability arises from a number of sources: instrument noise, different geometry acquisitions, shortcomings of the atmospheric correction, or fast changes in the soil brightness due to rain. Most of these factors are of little interest for the study of the vegetation, so they are usually considered "noise", and a number of techniques are employed to remove this signal from the clean data.

The most common practice in processing such data prior to attempting an interpretation is to apply some sort of low pass filter to the time series. The reason this is appropriate is that we have an expectation that the progression of a quantification of the vegetation should (in the absence of episodic events such as harvesting or fire) should be *smooth*, though we might not know *a priori* how smooth we expect it to be. In other words, there is an expectation that high frequency variability (the spiky nature of the time series) does not happen in vegetation. A low pass filter will remove the high frequency variability, leaving the smooth underlying time series. Recently, the Savitzky-Golay filter has been widely employed to smoooth timeseries of vegetation indices. The figure shows an example of the original and filtered versions of the output.



Note that the filtering relies on specifying its characteristics: what constitutes high frequency noise and how much does it need to be dampened by? The problem may be that we smooth too much or too little. The choice of parameters, in the absence of knowledge of the rate of change of things in the land surface is therefore problematic.

However, there are a number of issues to consider departures from simple smoothing of vegetation index data, although we will return to the concept of expectation of smoothness below:

1. Vegetation indices are quite limited in their information content.
2. Some of the non-smooth variation artefacts (such as acquisition geometry) can be explained by using suit-

able models. They are not noise, but useful signal.

3. A single univariate datum only gives a limited insight into the state of the land surface. Physical-based models can be used to tease out the contributions of different processes within the scene.
4. Thermal noise in the signal is usually well-characterised, and this knowledge has not yet been used.
5. Different processes that affect the signal of interest evolve at different rates, so different filtering mechanisms could compete. This is even more pronounced when using vegetation indices, where all the information is projected into a single scalar magnitude.
6. Without an understanding of the underlying signal, it is quite arbitrary to define appropriate filter characteristics for it.

These are just a few of the reasons why we may want to move away from simple filtering of vegetation indices for understanding the land surface, although VIs provide a powerful insight into (spatial and temporal) patterns of vegetation activity. We will introduce concepts of data assimilation and apply them to the problem of retrieving land surface parameters from optical sensors. For starters, we will still use a univariate time series, but will add increasing complexity to deal with different wavebands, the effect of the atmosphere and how we inject physical knowledge to the whole process.

2.3.2 Filtering and optimisation

We have seen that filtering a time series produces results that are 'easier' to interpret by reducing noise contributions to the signal. Additional insight is gained by noting that the desirable goal of removing high frequency components in the signal can be seen as equivalent of ensuring the signal is constrained to have relatively small changes between consecutive time steps. This is equivalent to saying that the best reconstruction of the original signal is the one where the timeseries' first (or higher order) derivatives are small. They are small, but not zero, as zero would imply a constant valued signal.

Relating this to ideas in (biogeochemical) process models, having 'minimal first order derivatives' is equivalent to a zero-order process model (i.e. the expectation is that the state tomorrow is the same as today, with some degree of uncertainty on that). Similarly, a first order process model (an expectation of constant rate of change) is the same as minimising second order derivatives.

If then we had an estimate of the signal's noise level, we could phrase the smoothing problem as a trade-off between "signal fidelity" and "expectation of smoothness". Graphically, the reconstructed signal would be the one that, while still staying within the errorbars of the measurements, gives the most smooth trajectory. Clearly, the observation and smoothness terms would need to be balanced properly. Further, we need to fit in uncertainty in the observations and in our prior smoothness expectation into the problem formulation.

A first approach would assume that each individual measurement in contaminated by independent Gaussian noise with a variance σ^2_{obs}. Stacking the uncertainties into a covariance matrix C_{obs}, we can write the contribution of the observation mismatch to our problem as J_{obs}:

$$J_{obs} = (1/2)(\mathbf{y_{obs}} - H(\mathbf{x}))^T(\mathbf{C_{obs}}^{-1} + \mathbf{C_H}^{-1})(\mathbf{y_{obs}} - H(\mathbf{x})),$$

where $\mathbf{x}$ is our estimate of the true value of the state (i.e. what we would like to estimate) and y_{obs} is an observation. The operator $H(\mathbf{x})$ maps from the space of $\mathbf{x}$ to the space of y_{obs}. In the general case, $\mathbf{C_H}^{-1}$ is the uncertainty in the result of the operator $H(\mathbf{x})$. We will ignore $\mathbf{C_H}^{-1}$ in the discussion below, as the current implementation of EOLDAS considers only $\mathbf{C_{obs}}^{-1}$, although it can be easily re-introduced into the expressions.

This is a very general expression, and we will see that it is this core equation that is at the heart of the DA system in EOLDAS.

In this particular case, the observation we are concerned with is our e.g. time series VI signal, so y_{obs} is a vector with elements equal to a stack of the VI data. We have already stated that C_{obs} is the uncertainty associated with this observation.

Since our state vector (i.e. what we want to estimate) is in the same space as the observations we can use an Identity operator for $H(\mathbf{x})$, i.e. $H(\mathbf{x}) = I(\mathbf{x})$. Another way of expressing this is to suggest that $\mathbf{x}$ is a *prior* estimate of the state variables that we wish to estimate, so we can call this a Prior Operator:

$$J_{prior} = (1/2)(\mathbf{y_{obs}} - I(\mathbf{x}))^T\mathbf{C_{obs}}^{-1}(\mathbf{y_{obs}} - I(\mathbf{x})),$$

We wish to find the values of $\mathbf{x}$ that give the minimum of J_{obs}, but the constraint with the identity operator is insufficient as the optimisation will result in the estimated state being the same as the input signal.

To express the condition of smoothness, we can encode that, for example, the signal change between consecutive time steps can be also be expressed through a Gaussian distribution with variance σ^2_{smooth}. The difference between time steps can be calculated by introducing a first order differential operator matrix $\mathbf{\Delta}$ with the following form:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$\mathbf{\Delta} = \begin{pmatrix} 1 & -1 & 0 & \cdots & 0 \\ 0 & 1 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & -1 \end{pmatrix}$$

Stacking again the variances into a a covariance matrix, we can now define J_{smooth} as

$$J_{smooth} = (1/2)\gamma^2(\mathbf{\Delta x})^T {\mathbf{C_{smooth}}}^{-1}(\mathbf{\Delta x}).$$

Here, γ is a term controlling the degree of smoothness: the higher it is the smoother the result will be. Since we can consider a constraint with a first order differential model as a zero-order process model (i.e. tomorrow is the same as today), we can phrase the inverse of γ^2 as the (squared) expectation of that process model: the expectation that, on average over the whole dataset, tomorrow will be the same as today. It is interesting to note that a zeroth order process model (a constraint of first order derivatives) provides a linear interpolation between sample points.

Note that since we can sometimes assume $\mathbf{C_{smooth}}$ to be diagonal, this particular functional can be written as

$$J_{smooth} = (1/2)\gamma^2 \sum_{i=1}^{N-1} \frac{(x^i - x^{i-1})^2}{\sigma^2_{smooth}}.$$

In other words, if the evolution of the signal is smooth (i.e., little change between timesteps, relative to σ_{smooth}), then J_{smooth} will be small. On the contrary, large and frequent changes between timesteps (again relative to σ_{smooth}) will result in a large contribution to J_{smooth}.

Note also that we can replace the differential operators above by arbitrary orders of differential operators. A second order constraint replaces $\mathbf{\Delta}$ in the above by $\mathbf{\Delta^2}$, for example and has the attraction of smoothly interpolating between sample points, rather than the piecewise linear result one gets from a first order derivative constraint.

We can extend the idea of smoothness expectation to other expectations that we might have about the signal prior to the data being available. These expectations will include signal bounds, smoothness over other different time scales, and perhaps an expectation of what the trajectory of the signal is, either from historical data or from the predictions of an ecosystem model. J_{smooth} states that we want a fit to the "model" predictions (in our example, the "model" basically states that the state does not change between timesteps), but we are quite happy to depart from this assumption if that provides a tighter fit to the observations. When we introduce σ_{smooth}, we introduce the concept that our expectation or state vector model has an error, and tightly fitting to it is thus undesirable.

So we have seen that we want to find an optimum in terms of J_{obs} and J_{smooth}. We define a function, J, the cost function, as the sum of both terms:

$$J = J_{smooth} + J_{obs}$$

The minimum of J will fit both the observations and our prior expectation of the model state. If we ignore J_{smooth}, we simply have a typical least squares problem, which may be indeterminate if the information content of the observations is not enough to constrain a unique solution (which is generally the case).

The extra constraint of J_{smooth} can be seen as a factor that 'simplifies' the problem, reducing the solution space, and making it 'easier' to solve. We can call this approach 'regularisation' as it helps to make the problem 'more regular' and tractable.

There is an obvious balance between our expectation of the signal (J_{smooth}) and our fidelity to the observations (J_{obs}), but we should not generally over-concerned with that: we do not want to over-fit the observations in any case because we understand them to have (random) error associated with them. The optimal solution will be dependent on the balance between σ_{obs} and σ_{smooth}. While the former may be available from apparatus calibration or ancillary measurements, the latter tends to be unknown or difficult to elucidate. We shall see what ways we have of identifying this parameter later on.

2.3.3 The problems the EOLDAS can solve

The EOLDAS prototype solves problems similar to the one expressed above. In general, these are termed 4DVAR problems with a weak constraint. Additionally, it is also possible to solve 4DVAR problems with strong constraint (this basically means that $\sigma_{smooth} = 0$, i.e., we trust that the solution lies fully in the space of our expectatons. We also note that we can add more terms to J to encode extra information about it. This is the core idea in a variational DA system: we can add constraints to the solution to express what we 'know' that might influence our estimate, provided we know something of the uncertainty.

So for example, we can encode that the solution should fit the observations, be smooth and follow a trend not too dissimilar to that given by a historical expectation. The problem is then posed as

$$J_{smooth}(\mathbf{x}) + J_{obs}(\mathbf{x}) + \cdots = \min(\mathbf{x})$$

This type of problems require the use of optimisation techniques, which can be computationally very expensive. To some degree, the use of the derivatives of the cost functionals alleviates this cost, so in the next sections, it will be assumed that the derivative of the function to minimise is known, either because it has been calculated explicitly or because a numerical approximation can be estimated.

Mathematically, the minumimum of J occurs when its derivative is zero. Since $\mathbf{x}$ generally has multiple elements (it is a vector) we desire the partial derivatives of J with respect to $\mathbf{x}$ to be zero.

2.4 A simple observation operator example, running in eoldas

2.4.1 Purpose of this section

This section of the user guide will take a simple example of combining two cost functions in a variational DA sense: an Identity (or Prior) operator and a smoothness (derivative) constraint. The examples used here are of filtering noisy time series of NDVI data.

As well as learning about these concepts if you are unfamiliar with them, this section will also take you through some examples of running the eoldas to solve this sort of problem. Towards the end of the section, we delve into writing some python code to make use of eoldas, and also mention other ways of interacting with it.

We learn that at the heart of eoldas is one or more configuration files that allow us to solve problems without the need for any code writing (though you can if you want to!).

Introducing the observation operator concept

In the previous sections, we have assumed that the state of the land surface can be observed directly, and that these observations are only limited by noise in their acquisition. In general, the state of the surface and the observations will be different entities. For example, the state of the surface may include parameters such as leaf area index, chlorophyll concentration or soil moisture, whereas the observations may be of directional surface reflectance, top of atmosphere radiance or microwave temperatures. The link between these magnitudes is the observation operator, which maps the land surface parameters into quantities that are measured by a sensor. Many types of observation operator are available: from the statistical to the physics based. The simplest case, however, is the identity operator. In fact, we have already used it, as we have assumed that the observations are just direct measurements of the state vector.

A simple data assimilation example using an identity observation operator

The simplest observation operator is the identity observation operator, where the observations are identical to the state vector components. We can see the use of this operator as a way of optimally smoothing univariate timeseries, for example NDVI. Additionally, the use of a DA system allows to interpolate where data points are not available. The following demonstrates how the EOLDAS prototype can be used for this task, and also allows us to explore the use of the weak assimilation paradigm conveniently.

The main way to run EOLDAS is via one or more configuration files. This is partly to make sure that a record exists of a particular experimental setup and partly to allow flexible running of the system without the need for further coiding by the user (unless he/she wants to add new classes or methods).

Here is the start of the configuration file that we are going to use:

```
# An EOLDAS confioguration file for
# a simple Identity operator and a regulariser

[parameter]
location = ['time']
limits = [[1,365,1]]
names = gamma_time,NDVI
solve = 0,1
datatypes = x

[parameter.result]
filename = output/Identity/NDVI_Identity.params
format = 'PARAMETERS'
help_filename='Set the output state filename'

[parameter.x]
datatype = x
names = $parameter.names
default = 200,0
help_default="Set the default values of the states"
apply_grid = True
sd = [1.]*len($parameter.names)
bounds =  [[0.000001,1000000],[-1,1]]
```

This sets up the section `[parameter]`, which described the state variables within EOLDAS.

The section must contain fields describing the locational information of the data to be used as well as the names of the state variables we will consider. In this case, we have two state variables `gamma_time` and `NDVI` that we wish to estimate over the (time) range 1 to 365 (inclusive) in steps of 1 (day).

The subsection parameter.solve gives a list of codes indicating whether we wish to solve for each parameter or not.

A code of 0 tells the EOLDAS not to solve, i.e. just to use the data that are read in or any other default values.

A code of 1 tells EOLDAS to solve for that state variable at all locations (times here).

A code of 2 tells EOLDAS to solve for the state variable, but to maintain a single value over all locations.

The section `parameter.result` gives information on any output file name and format for the state.

Finally in this section, the field `parameter.x` sets up data and conditions for the state vector `x`. Remember that it is this state vector `x` that we will solve for in the EOLDAS. Here, we specify that it is of datatype `x` (i.e. the `x` state vector), that it has the same names as we set up in `parameter.names`, that default values to be assigned are 25 and 0 for the two variables respectively. If any data are read in, these override the default values, but in this case, we simply start with the defaults.

The text `help_default` allows the field `parameter.x.default` to be set from the command line with the option `--parameter.x.default=a,b`.

The flag `apply_grid`, which is the default, tells the EOLDAS to produce the state vector on a grid over the bounds defined in parameter.limits.

Finally, we define default uncertainty information for the state vector (this does not directly affect the running of the EOLDAS (`parameter.x.sd`), and define the bounds for each state vector (use None if no bound is to be used). Here, we set the lower bound as 0 and the upper bound as 1 for both parameters.

The next section sets up general conditions:

```
[general]
doplot=True
help_do_plot='do plotting'
```

In this case, we set a flag to do plotting when the results are written out. Plotting will use the filenames in any state variable section e.g. `parameter.result.filename` to generate a series of plots with filenames the same as this data filename. We will see some examples later.

The next section sets up the operators that we want to define here.

```
[operator]
modelt.name=DModel_Operator
modelt.datatypes = x
obs.name=Operator
obs.datatypes = x,y
```

Here, we define two operators, `DModel_Operator` and `Operator`. These names refer directly to python classes for the operators in EOLDAS. The base class is 'Operator' which implements the Identity operator. All other classes are derived from this. The differential operator works only on the x state vector, which is equlivalent to defining $\mathbf{y_{obs}} = 0$. The operator 'Operator' access both x and y data if it to act as a Prior constraint, so we set up x and y datatypes.

Next we set the details of these operators. First, the differential operator:

```
[operator.modelt.x]
names = $parameter.names
sd = [1.0]*len($operator.modelt.x.names)
datatype = x

[operator.modelt.rt_model]
model_order=1
help_model_order='The differential model order'
wraparound=periodic,365
```

where we specify which state vector elements this operator has access to (all of those in parameter.names here) and set up the default uncertainty and datatype.

We then set the parameters specific to the 'model' $H(\mathbf{x})$, in this case the order of the differential model (2 here) and the edge conditions (periodic, with a period of 200 (days)).

Finally, we set up the operator `operator.obs`, specifically, parameters for its x and y state vectors.

```
[operator.obs.x]
names = $parameter.names[1:]
sd = [1.0]*len($operator.obs.x.names)
help_sd='Set the observation sd'
datatype = x

[operator.obs.y]
control = [mask]
names = $parameter.names[1:]
sd = [0.15]*len($operator.obs.x.names)
help_sd="set the sd for the observations"
datatype = y
state = data/Identity/random_ndvi1.dat
help_state = "Set the y state vector"

[operator.obs.y.result]
filename = output/Identity/NDVI_fwd.params
```

specifying default uncertainty information, data types and any required output files. Here, we wish to write out the results in `operator.obs.y`, so we specify a filename for this. Again, we see the use of a 'help' variable, which here allows `operator.obs.y.result.filename` to be set from the command line. This interfacing

to the command line means that a single configuration file can generally serve for multiple experiments and the user does not need to keep generating new ones.

The main program can be accessed in various ways. One way is to write some front end code that calls the eoldas python code.

An example is solve_eoldas_identity.py that includes three sections:

First, some code to generate a synthetic dataset.

```
import pdb
import numpy as np

def create_data ( n_per=4, noise=0.15, obs_off=0.33, \
                                        window_size=0, order=4):
    """
    Create synthetic "NDVI-like" data for a fictitious time series. We return
    the original data, noisy data (using IID Gaussian noise), the QA flag as well
    as the time axis.

    Missing observations are simulated by drawing a random number between 0 and 1
    and checking against obs_off.

    Parameters
    ----------
    n_per : integer
    Observation periodicity. By default, assumes every 8 days

    noise : float
    The noise standard deviation. By default, 0.15

    obs_off : float
    The threshold to decide on missing observations in the time series.

    window_size : integer, odd
    window size for savitzky_golay filtering. A large window size will lead
    to larger data gaps by correlating the noise. Set to zero by default
    which applies no smoothing.

    order : integer
    order of the savitzky_golay filter. By default 4.

    """
    from savitzky_golay import savitzky_golay
    import numpy as np

    doys  = np.arange ( 1, 365+1, n_per)
    ndvi_clean =  np.clip(np.sin((doys-1)/72.), 0,1)
    ndvi =  np.clip(np.sin(doys/72.), 0,1)
    # add Gaussian noise of sd noise
    ndvi = np.random.normal(ndvi,noise,ndvi.shape[0])

    # set the qa flags for each sample to 1 (good data)
    qa_flag = np.ones_like ( ndvi).astype( np.int32 )
    passer = np.random.rand( ndvi.shape[0])
    if window_size >0:
        # force odd
        window_size = 2*(window_size/2)+1
        passer = savitzky_golay(passer, window_size=window_size, order=order)
    # assign a proportion of the qa to 0 from an ordering of the smoothed
    # random numbers
    qa_flag[np.argsort(passer)[:passer.size * obs_off]]  = 0

    return ( doys, ndvi_clean, ndvi, qa_flag )
```

Here, we generate some NDVI data which has a trajectory of a sine wave for the first half of the year and is flat at zero for the second half. The sampling is controlled by n_per and obs_off. The parameter obs_off randomly removes a proportion of the data. If window_size and order are set then a savitzky_golay filter is used to induce correlation in the timing of the samples that are removed from thie dataset (qa=0). This mimics what we practically have in Optical Earth Observation with temporal correlation in cloud cover.

The next section calculates the 'ideal' value of γ by calculating the root mean squared deviation of the original dataset. We use this ideal gamma here for to demonstrate the physical meaning of the gamma value, though in practice this would be unknown. This section also writes the dataset to a temporary file in 'BRDF' format.

All of this so far standard python coding, though such datasets could be generated in many other ways. The final section interfaces to the top level of the eoldas code, which is what is of immediate concern in this tutorial.

```
import sys,tempfile
this = sys.argv[0]
import  eoldas

# SD of noise
noise=0.15
# nominal sampling period
n_per=7
# proportion of missing samples
obs_off=0.33
# order of differential model (integer)
model_order=1
# sgwindow is larger to create larger data gaps
sgwindow=10

# set up data for this experiment
file, ideal_gamma,doys,ndvi_clean,ndvi,qa_flag  = \
            prepare(noise=noise,n_per=n_per,\
                    obs_off=obs_off,model_order=model_order,\
                    sgwindow=sgwindow)

# set gamma to less than the the theoretical value
gamma = ideal_gamma*0.33

# set op file names
xfile = 'output/Identity/NDVI_Identity.params'
yfile = 'output/Identity/NDVI_fwd.params'

# initialise optuions for DA overriding any in config files
cmd = 'eoldas ' +  \
            ' --conf=config_files/eoldas_config.conf --conf=config_files/Identity.conf ' + \
            ' --logfile=mylogs/Identity.log ' + \
            ' --calc_posterior_unc ' + \
            ' --parameter.solve=0,1 ' + \
            ' --parameter.result.filename=%s '%xfile +\
            ' --parameter.x.default=%f,0.0 '%(gamma) + \
            ' --operator.obs.y.result.filename=%s'%yfile +\
            ' --operator.obs.y.state=%s'%file+\
            ' --operator.modelt.rt_model.model_order=%d '%model_order

# initialise eoldas
self = eoldas.eoldas(cmd)
# solve DA
self.solve(write=True)
```

The first part of the code extends the system path for where it searches for libraries. This is done relative to where solve_eoldas_identity.py is (in the bin directory of the distribution). After that, we set up values for the parameters for generating the synthetic dataset.

The interface to the eoldas here is mainly to make a string with a number of flags. The most important flag is `--conf=config_files/Identity.conf` which specifies the configuration file for this experiment.

In addition, `--conf=eoldas_config.conf` is given, which specifies a system default configuration file, `eoldas_config.conf`.

So, the simplest 'top level' interface to eoldas from python code involves:

```
    gamma = ideal_gamma*0.33

    # set op file names
    xfile = 'output/Identity/NDVI_Identity.params'
    yfile = 'output/Identity/NDVI_fwd.params'

    # initialise optuions for DA overriding any in config files
    cmd = 'eoldas ' +  \
                ' --conf=config_files/eoldas_config.conf --conf=config_files/Identity.conf ' + \
                ' --logfile=mylogs/Identity.log ' + \
                ' --calc_posterior_unc ' + \
                ' --parameter.solve=0,1 ' + \
                ' --parameter.result.filename=%s '%xfile +\
                ' --parameter.x.default=%f,0.0 '%(gamma) + \
                ' --operator.obs.y.result.filename=%s'%yfile +\
                ' --operator.obs.y.state=%s'%file+\
                ' --operator.modelt.rt_model.model_order=%d '%model_order
```

and

```
                ' --operator.modelt.rt_model.model_order=%d '%model_order

    # initialise eoldas
    self = eoldas.eoldas(cmd)
    # solve DA
    self.solve(write=True)
```

where we set up the text string, initiate the eoldas object (`eoldas.eoldas`) and then call the `eoldas.solve()` method.

This command string is clearly of some importance. The flags `--logfile=mylogs/Identity.log` and `--calc_posterior_unc` correspond to items in the `[general]` section of the configuration file. Here, `eoldas_config.conf` contains the lines:

```
[general]
datadir = ., data
help_datadir ="Specify where the data and or conf files are"
here = os.getcwdu()
grid = True
is_spectral = True
calc_posterior_unc=False
help_calc_posterior_unc ="Switch to calculate the posterior uncertainty"
write_results=True
help_write_results="Flag to make eoldas write its results to files"
init_test=False
help_init_test="Flag to make eoldas run a test of the cost functions before proceeding with DA"
doplot=False
plotmod=100000000
plotmovie=False

[general.optimisation]
# These are the default values
iprint=1
gtol=1e-3
help_gtol = 'set the relative tolerance for termination'
maxIter=1e4
maxFunEvals=2e4
plot=0
# see http://openopt.org/NLP#Box-bound_constrained
```

```
solverfn=scipy_lbfgsb
randomise=False
help_randomise='Randomise the starting point'
```

To understand how e.g the flag `--calc_posterior_unc` can be used we can look at:

```
calc_posterior_unc=False
help_calc_posterior_unc ="Switch to calculate the posterior uncertainty"
```

where we set the default value of the item (`False`) and also have a 'help' statement which allows this value to be overridden. You shouldn't normally need to change things in the system configuration file `eoldas_config.conf`. This flag controls whether we calculate the posterior iuncertainty or not. The default is False because it can be quite computationally expensive and is often best done in a post processing step.

The flag `--operator.obs.y.state=filename` refers specifically to the section `operator.obs.y.state` of the configuration, which is something we have set up in `Identity.conf` with a 'help' field, so we can override the default value set in the configuration file.

```
[operator.obs.y]
control = [mask]
names = $parameter.names[1:]
sd = [0.15]*len($operator.obs.x.names)
help_sd="set the sd for the observations"
datatype = y
```

If we run `solve_eoldas_identity.py`, it sends logging information to `mylogs/Identity.log` and reports (to the stderr) the progress of the optimisation, 'f' being the total of all of the J terms for this configuration. It should converge to a solution within some tens of iterations and result in a final value of J of around 1800. We set the name of the logfile in `solve_eoldas_identity.py`:

```
    if window_size >0:
```

There is a lot of detail in the log file about exactly what value terms are set to and the progress of the eoldas. It also contains information on the individual J terms:

```
2012-06-14 15:15:57,455 - eoldas.solver.eoldas.solver-obs-x - INFO -      J   = 293.213607
2012-06-14 15:15:57,456 - eoldas.solver.eoldas.solver-modelt-x - INFO -      J   = 34.354830
2012-06-14 15:15:57,457 - eoldas.solver.eoldas.solver-obs-x - INFO -      J   = 196.657803
2012-06-14 15:15:57,458 - eoldas.solver.eoldas.solver-modelt-x - INFO -      J   = 87.225910
2012-06-14 15:15:57,459 - eoldas.solver.eoldas.solver-obs-x - INFO -      J   = 112.026943
2012-06-14 15:15:57,461 - eoldas.solver.eoldas.solver-modelt-x - INFO -      J   = 113.479088
2012-06-14 15:15:57,462 - eoldas.solver.eoldas.solver-obs-x - INFO -      J   = 78.729997
2012-06-14 15:15:57,463 - eoldas.solver.eoldas.solver-modelt-x - INFO -      J   = 93.761052
2012-06-14 15:15:57,464 - eoldas.solver.eoldas.solver-obs-x - INFO -      J   = 43.172989
2012-06-14 15:15:57,466 - eoldas.solver.eoldas.solver-modelt-x - INFO -      J   = 99.078418
2012-06-14 15:15:57,467 - eoldas.solver.eoldas.solver-obs-x - INFO -      J   = 40.564246
2012-06-14 15:15:57,468 - eoldas.solver.eoldas.solver-modelt-x - INFO -      J   = 74.685441
2012-06-14 15:15:57,469 - eoldas.solver.eoldas.solver-obs-x - INFO -      J   = 39.829009
2012-06-14 15:15:57,470 - eoldas.solver.eoldas.solver-modelt-x - INFO -      J   = 63.881006
2012-06-14 15:15:57,471 - eoldas.solver.eoldas.solver-obs-x - INFO -      J   = 27.079490
```

A log file is important to the running of eoldas, as the processing can take quite some time for some problems.

The state vector results will be written to output/Identity/NDVI_Identity.params because we first specified this in Identity.conf:

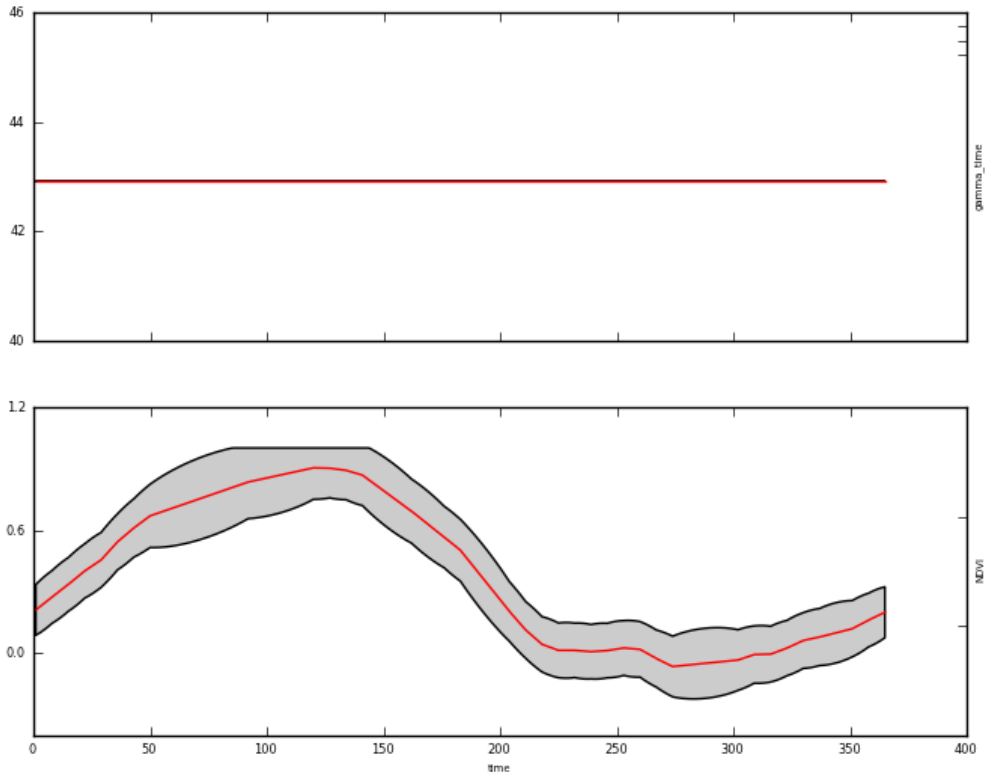
```
[parameter.result]
filename = output/Identity/NDVI_Identity.params
format = 'PARAMETERS'
```

(actually, since we also set the comamnd `' --operator.obs.y.result.filename=%s'%yfile` in the code we have written, this will override what is in the parameter file).

The output file format is a general 'PARAMETERS' format that should look something like this:

The first line of the file is a header statement that describes the location fields ('time' here) and state variables (gamma_time and NDVI here) and subsequent lines give the values for those fields. This format can be used for input data, but it cannot at present be used to define a full input covariance matrix, only the standard deviation for each observation as in this example.

A graph of the result is in `output/Identity/NDVI_Identity.params.plot.x.png`:



This result is quite interesting for understanding how our DA system works: The plot shows the mean of the estimate of the NDVI state vector in the lower panel (the upper panel shows the value of gamma_time which we did not solve for) as a red line as we solve for the state every day (of 365 days). The 95% confidence interval is shown shaded in grey. We will show this below in a more refined plot of the results.

We also specified the 'y' data to be written out (in Identity.conf):

```
help_state = "Set the y state vector"

[operator.obs.y.result]
filename = output/Identity/NDVI_fwd.params
```

so this goes to a file `output/Identity/NDVI_fwd.params` unless the flag `--operator.obs.y.result.filename=somethingElse.dat` is used. The format of this file is the same as above, but it shows the retrieved NDVI data since this reports $H(x)$.

```
#PARAMETERS time mask NDVI sd-NDVI
   1.000000    1.000000    0.211816    0.000000
   8.000000    1.000000    0.275085    0.000000
  15.000000    1.000000    0.336462    0.000000
  22.000000    1.000000    0.401630    0.000000
  29.000000    1.000000    0.455163    0.000000
  36.000000    1.000000    0.542784    0.000000
  43.000000    1.000000    0.610468    0.000000
  50.000000    1.000000    0.669554    0.000000
```

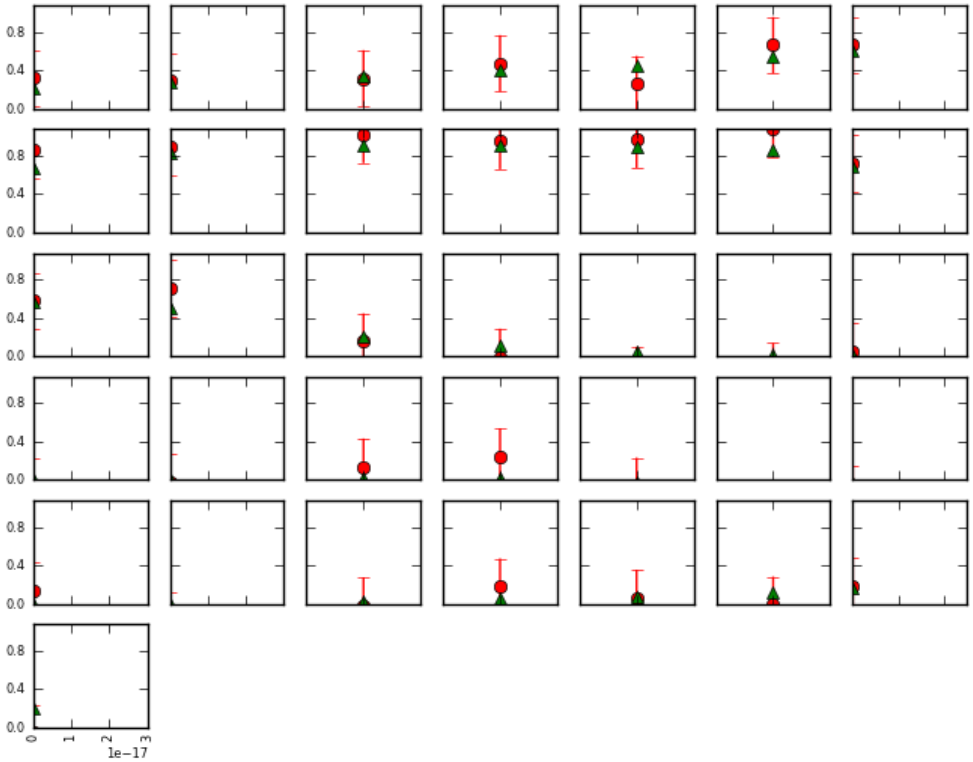
```
  92.000000    1.000000    0.834135    0.000000
```

The original dataset is output form convenience in the same format as the y state, being `output/Identity/NDVI_fwd.params_orig` here:

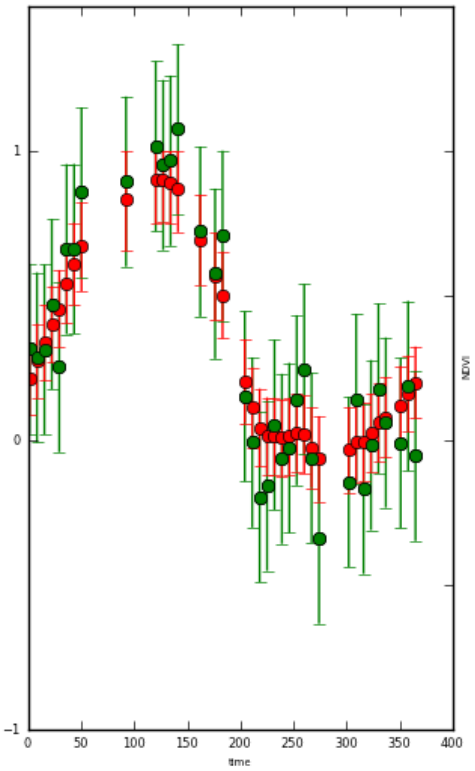
```
#PARAMETERS time mask NDVI sd-NDVI
   1.000000    1.000000    0.315700    0.150000
   8.000000    1.000000    0.286300    0.150000
  15.000000    1.000000    0.313996    0.150000
  22.000000    1.000000    0.470529    0.150000
  29.000000    1.000000    0.253425    0.150000
  36.000000    1.000000    0.660812    0.150000
  43.000000    1.000000    0.661460    0.150000
  50.000000    1.000000    0.856904    0.150000
  92.000000    1.000000    0.894455    0.150000
```

Various other graphics are output for a 'y' state:

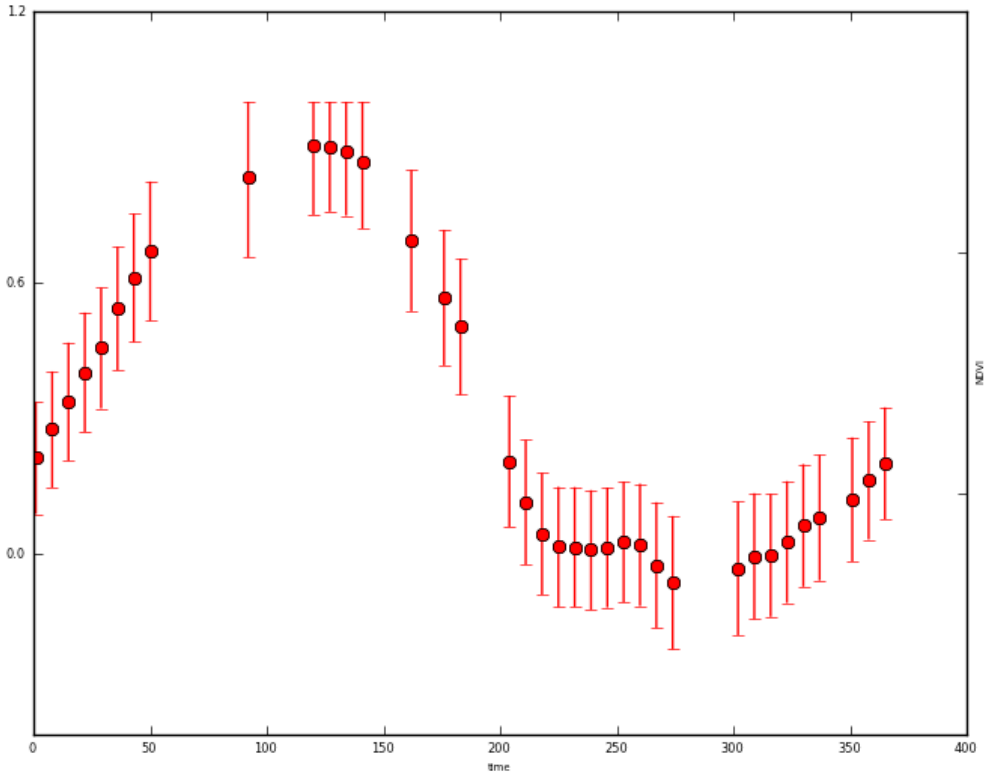
A plot at each location, in output/Identity/NDVI_fwd.params.plot.y.png (not of much relevance in this example):



A plot of the observations and modelled values as a function of location (the observations are the green dots) in output/Identity/NDVI_fwd.params.plot.y2.png:



and a plot of the x state vector associated with this operator (NDVI here).



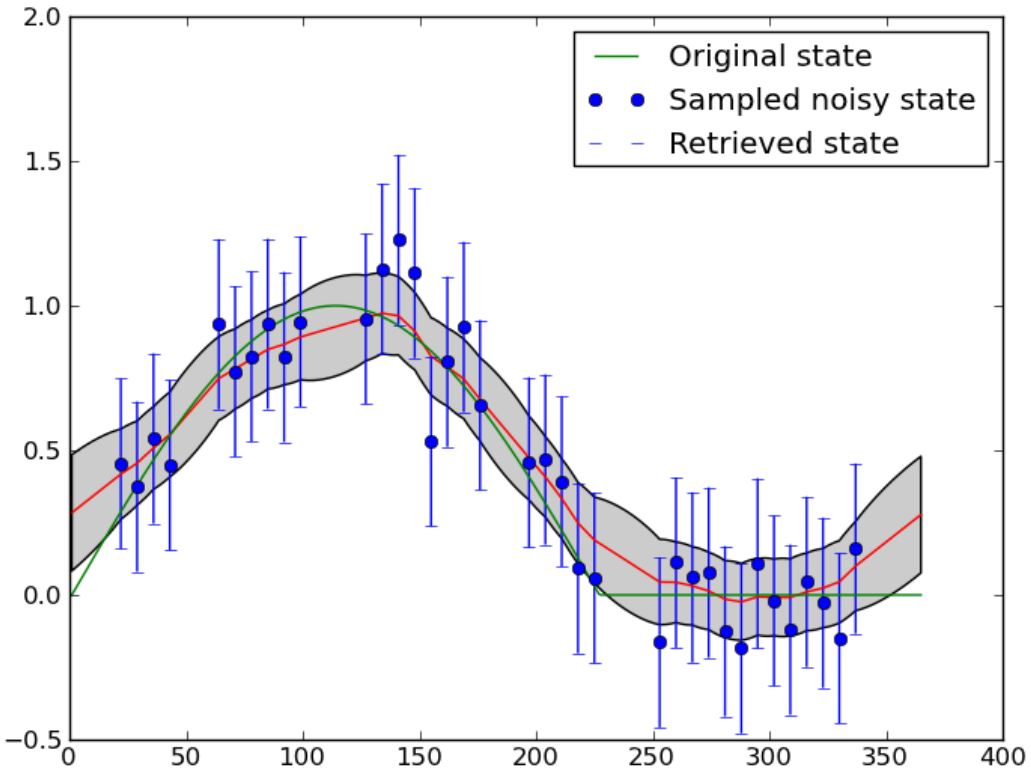
2.4.2 Example plotting data from the output files

The above plots are automatically generated by eoldas provided general.doplot is True but these are intended as quicklooks, and users are likely to want to form their own plots.

An example of this is implemented in example1plot.py:

```
#!/usr/bin/env python
import numpy as np
import pylab as plt
#
# Some plotting of the synthetic and retrieved data

# generate the clean data
doys  = np.arange ( 1, 365+1, 1)
ndvi_clean =  np.clip(np.sin((doys-1)/72.), 0,1)

# read the state file
state = open('output/Identity/NDVI_Identity.params').readlines()
sdata = np.array([np.array(i.split()).astype(float) for i in state[1:]])
sNdvi = sdata[:,2]
sdNdvi = sdata[:,4]

# noisy sample data
noisy = open('output/Identity/NDVI_fwd.params_orig').readlines()
ndata = np.array([np.array(i.split()).astype(float) for i in noisy[1:]])
nNdvi = ndata[:,2]
sdnNdvi = ndata[:,3]
ndoys = ndata[:,0]
```

```
# retrieved
plt.fill_between(doys,y1=sNdvi-1.96*sdNdvi,y2=sNdvi+1.96*sdNdvi,facecolor='0.8')
p1 = plt.plot(doys,sNdvi,'r')
# original
p2 = plt.plot(doys,ndvi_clean,'g')
# samples used
p3 = plt.errorbar(ndoys,nNdvi,yerr=sdnNdvi*1.96,fmt='bo')

plt.legend([p2,p3,p1],['Original state','Sampled noisy state','Retrieved state'])

#plt.show()
plt.savefig('images/example1plot.png')
```



We can see that smoothers of this sort have some difficulty maintaining sudden changes, although this is quite challenging here given the level of noise amd the rather large data gaps.

More importantly, we have used the smoother to interpolate over the missing observations and to reduce uncertainty.

If we inspect the file `output/Identity/NDVI_Identity.params <output/Identity/NDVI_Identity.params>`:

```
#PARAMETERS time gamma_time NDVI sd-gamma_time sd-NDVI
   1.000000   42.912561    0.211816    0.000000    0.063006
   2.000000   42.912561    0.220854    0.000000    0.064164
   3.000000   42.912561    0.229893    0.000000    0.065042
   4.000000   42.912561    0.238931    0.000000    0.065651
   5.000000   42.912561    0.247970    0.000000    0.066000
   6.000000   42.912561    0.257008    0.000000    0.066092
   7.000000   42.912561    0.266046    0.000000    0.065928
   8.000000   42.912561    0.275085    0.000000    0.065507
   9.000000   42.912561    0.283853    0.000000    0.066408
```

alongside the original data `output/Identity/NDVI_fwd.params_orig <output/Identity/NDVI_Identity.params>`:

```
#PARAMETERS time mask NDVI sd-NDVI
   1.000000    1.000000    0.315700    0.150000
   8.000000    1.000000    0.286300    0.150000
  15.000000    1.000000    0.313996    0.150000
  22.000000    1.000000    0.470529    0.150000
  29.000000    1.000000    0.253425    0.150000
  36.000000    1.000000    0.660812    0.150000
  43.000000    1.000000    0.661460    0.150000
  50.000000    1.000000    0.856904    0.150000
  92.000000    1.000000    0.894455    0.150000
```

We note that the original state here (green line) lies entirely within the 95% CI. The error reduction has been of the order of 2.2 (compare sd-NDVI after the DA with that prior to it). We can see that the uncertainty at the datapoints has been reduced from 0.15 (that of the input data) to typically around 0.065. This grows slightly (to around 0.10) when the data gaps are large.

The impact of 'filtering' in this way (optimising a fit of the model to the observations) is to smooth the data and reduce uncertainty in the output. The reduction in uncertainty is related to the amount of smoothing that we apply. The second effect is to interpolate between observations, with uncertainty growing where we have no observations.

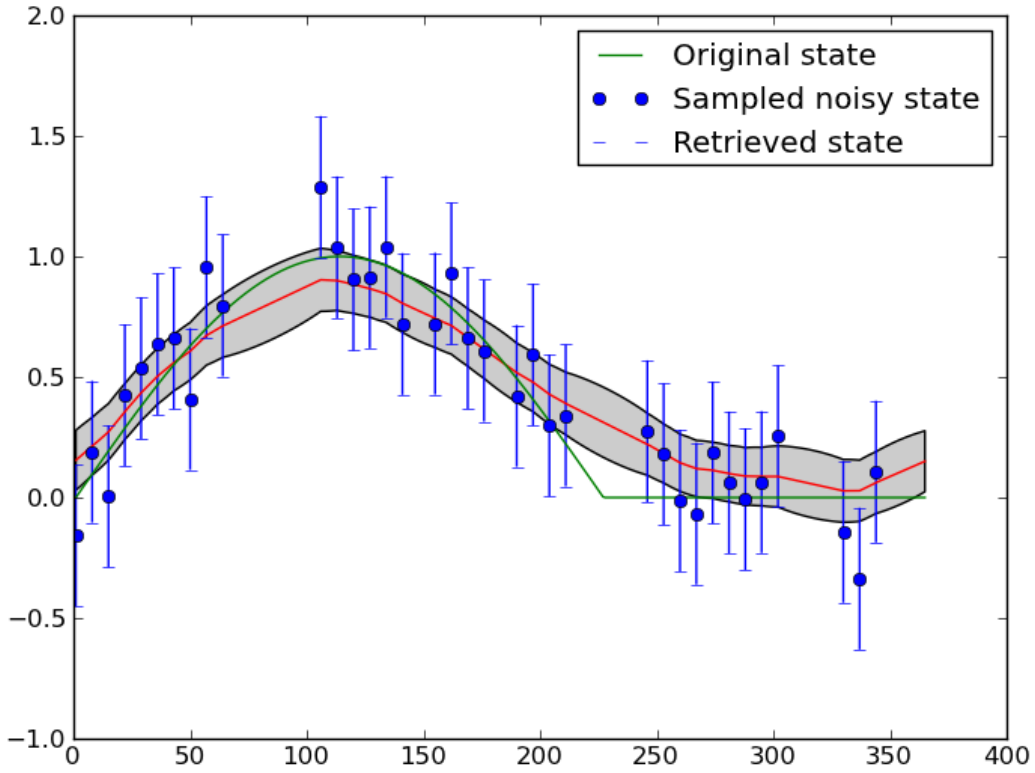
You might try changing the value of gamma used and seeing the effect on the results.

We could do this by modifying a few lines of `solve_eoldas_identity.py` to produce `solve_eoldas_identity1.py`:

```
if __name__ == "__main__":
    import sys,tempfile
    this = sys.argv[0]
    import eoldas

    # SD of noise
    noise=0.15
    # nominal sampling period
    n_per=7
    # proportion of missing samples
    obs_off=0.33
    # order of differential model (integer)
    model_order=1
    # sgwindow is larger to create larger data gaps
    sgwindow=10

    # set up data for this experiment
    file, ideal_gamma,doys,ndvi_clean,ndvi,qa_flag  = \
                prepare(noise=noise,n_per=n_per,\
                        obs_off=obs_off,model_order=model_order,\
                        sgwindow=sgwindow)

    # set gamma to less than the the theoretical value
    gamma = ideal_gamma*0.45

    # set op file names
    xfile = 'output/Identity/NDVI_Identity1.params'
    yfile = 'output/Identity/NDVI_fwd1.params'

    # initialise optuions for DA overriding any in config files
```

which writes out to `output/Identity/NDVI_Identity1.params` and `output/Identity/NDVI_fwd1.params` and has a gamma value that is 0.45/0.33 of that previously used.

Now, plotting this using `example1plot1.py`:



This is possibly a better result, but in fact what we see is further limitation of the model that we have chosen here: we enforce wraparound (i.e. the NDVI at day 1 is expected to be the same as at day 365) and we enforce smoothness (so as we increase the gamma, the smoothness, we over-smooth at the sudden change that occurs half way through the year).

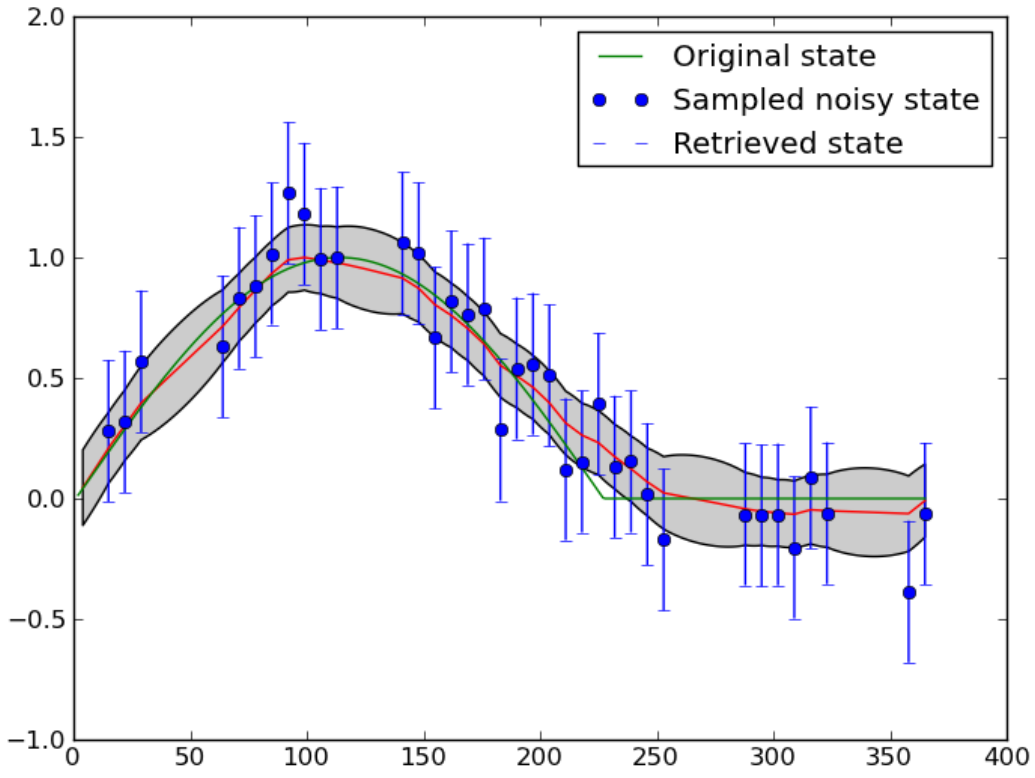
We could remove the wraparound condition, but in practice, it is better simply to weaken this constraint. We have done this in `solve_eoldas_identity2.py`:

```
if __name__ == "__main__":
    import sys,tempfile
    this = sys.argv[0]
    import eoldas

    # SD of noise
    noise=0.15
    # nominal sampling period
    n_per=7
    # proportion of missing samples
    obs_off=0.33
    # order of differential model (integer)
    model_order=1
    # sgwindow is larger to create larger data gaps
    sgwindow=10

    # set up data for this experiment
    file, ideal_gamma,doys,ndvi_clean,ndvi,qa_flag  = \
                prepare(noise=noise,n_per=n_per,\
                        obs_off=obs_off,model_order=model_order,\
                        sgwindow=sgwindow)

    # set gamma to less than the the theoretical value
    gamma = ideal_gamma*0.33
```

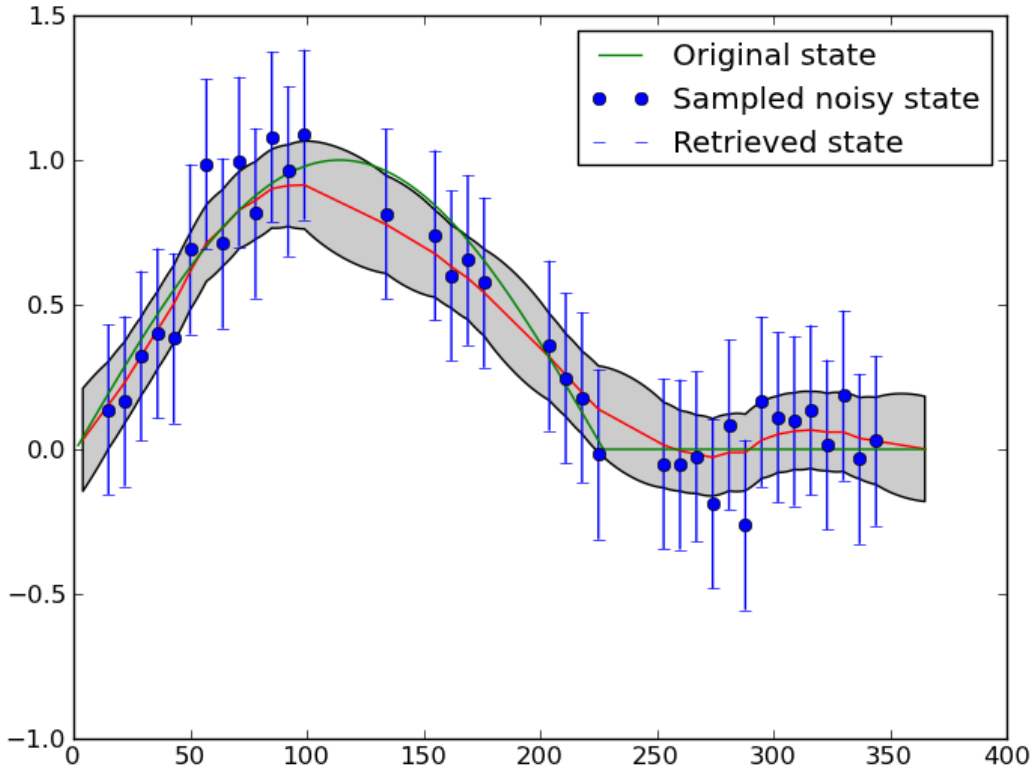
```
# set op file names
xfile = 'output/Identity/NDVI_Identity2.params'
yfile = 'output/Identity/NDVI_fwd2.params'

# initialise optuions for DA overriding any in config files
cmd = 'eoldas ' +  \
            ' --conf=config_files/eoldas_config.conf --conf=config_files/Identity.conf ' + \
            ' --logfile=mylogs/Identity.log ' + \
            ' --calc_posterior_unc ' + \
            ' --parameter.solve=0,1 ' + \
            ' --parameter.result.filename=%s '%xfile +\
            ' --parameter.x.default=%f,0.0 '%(gamma) + \
```

which write out to `output/Identity/NDVI_Identity2.params` and `output/Identity/NDVI_fwd2.params` and has the same (higher) gamma value used above.

Now, plotting this using `example1plot2.py`:



If you re-run these scripts several times, so that you see different configurations for the temporal sampling, you will notice that the interpolation sometimes behaves well over the entire dataset (for some given gamma) and sometimes doesn't. For example:



2.4.3 Interfacing a little more deeply with the eoldas code

Whilst considering writing wrapper codes around eoldas functionality and outputs it is instructive to explore some of the data structure available.

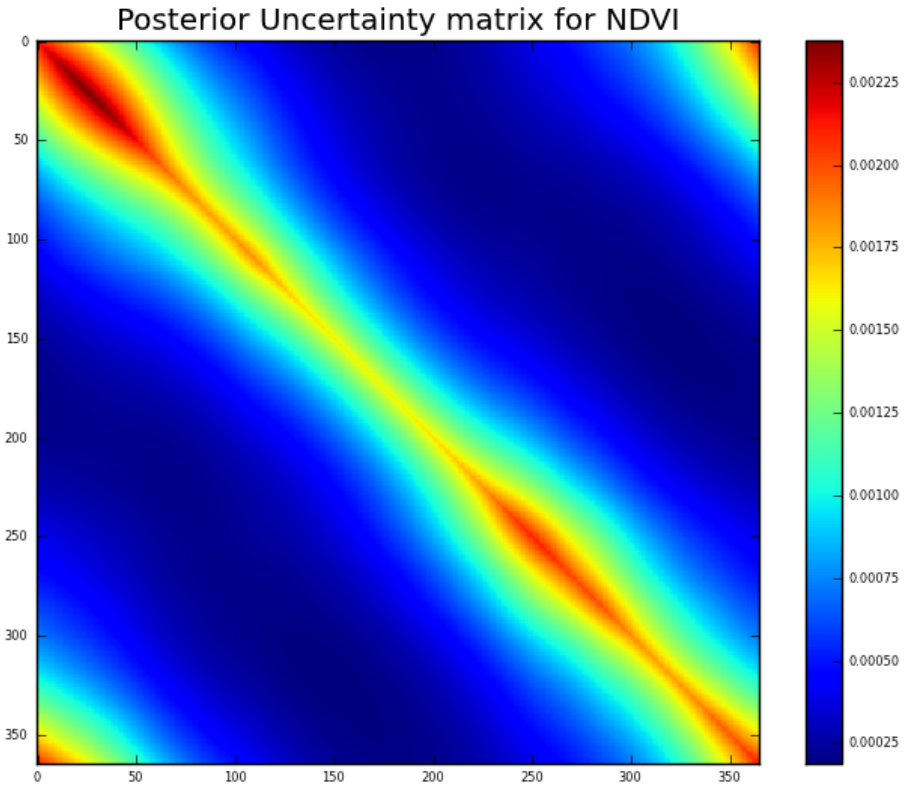
We can re-use the example solve_eoldas_identity.py developed above and access some of the data structure as shown in solve_eoldas_identity_a.py.

```
#!/usr/bin/env python
import pdb
import numpy as np

if __name__ == "__main__":
    import sys,tempfile
    this = sys.argv[0]
    import  eoldas
    from solve_eoldas_identity import *
    # import the setup methods from solve_eoldas_identity
    import pylab as plt

    # SD of noise
    noise=0.15
    # nominal sampling period
    n_per=7
    # proportion of missing samples
    obs_off=0.33
    # order of differential model (integer)
    model_order=1
    # sgwindow is larger to create larger data gaps
    sgwindow=10
```

```
# set up data for this experiment
file, ideal_gamma,doys,ndvi_clean,ndvi,qa_flag  = \
            prepare(noise=noise,n_per=n_per,\
                    obs_off=obs_off,model_order=model_order,\
                    sgwindow=sgwindow)

# set gamma to thge theoretical value
gamma = ideal_gamma

# set op file names
xfile = 'output/Identity/NDVI_Identity_a.params'
yfile = 'output/Identity/NDVI_fwd_a.params'

# initialise options for DA overriding any in config files
# make sure we use some different output file names to othe scripts
cmd = 'eoldas ' +  \
            ' --conf=config_files/eoldas_config.conf --conf=config_files/Identity.conf ' + \
            ' --logfile=mylogs/Identity.log ' + \
            ' --calc_posterior_unc ' + \
            ' --parameter.solve=0,1 ' + \
            ' --parameter.result.filename=%s '%xfile +\
            ' --parameter.x.default=%f,0.0 '%(gamma) + \
            ' --operator.obs.y.result.filename=%s'%yfile +\
            ' --operator.obs.y.state=%s'%file+\
            ' --operator.modelt.rt_model.model_order=%d '%model_order

# initialise eoldas
self = eoldas.eoldas(cmd)
# solve DA
self.solve(write=True)

# now pull some data out of the eoldas

# the 'root' of the DA data structure is in self.solver.root

root = self.solver.root

# The state vector data are stored in root.x
# with ancillary information in root.x_meta
# so the state vector is e.g. in root.x.state
# and the names are in root.x_meta.state

state_names = root.x_meta.state
state = root.x.state

# The sd representation of the posterior is in root.x.sd
# This is all set up in eoldas_Solver.py
# All storage is of type ParamStorage, an extended
# dictionary structure. You can explore it interactively
# with e.g. root.dict().keys() or self.keys() since
# self here is a straight dictionary
sd = root.x.sd

# The full inverse Hessian is in self.solver.Ihessian
Ihessian = self.solver.Ihessian

# A mask to reduce this to only the state variables
# being targeted (solve == 1) is through a call to:
NDVI_Ihessian = self.solver.unloader(None,Ihessian,M=True)

# This is of shape (365,365) here.
# so now lets produce an image of it
```

```
# to visualise the structure
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.imshow(NDVI_Ihessian,interpolation='nearest')
ax.set_title('Posterior Uncertainty matrix for NDVI')
# Add colorbar, make sure to specify
# tick locations to match desired ticklabels
cbar = fig.colorbar(cax) #), ticks=[-1, 0, 1])
#cbar.ax.set_yticklabels(['< -1', '0', '> 1'])# vertically oriented colorbar
# see http://matplotlib.sourceforge.net/plot_directive/mpl_examples/pylab_examples/colorbar_t
# save it
plt.savefig('output/IHessianNDVI_expt1.png')
```

The comments in the code should be self explanatory and anyone interested in delving much further into the eoldas codes should see the full class documentation. Here, we can see at least how to access the posterior estimate of the state and its uncertainty. We write out the uncertainty to an image using matplotlib (pylab):



Now, this is a very interesting figure for understanding how these multiple constraints are interacting. The observation uncertainty is just described by standard deviation, so lies along the leading diagonal of the *a priori* uncertainty. Further, it only exists where there are data points. The impact of applying the (regularisation) model constraint is to reduce the uncertainty at the observation points as we would expect. We can see a 'sausage' pattern from above in this figure quite clearly. The 'pinch points' are when the sample points are dense. Where there are large data gaps (from our simulated cloud impacts here) the uncertainty is higher, but it is 'spread out' from the leading diagonal by applying temporal covariance. The impact of the filtering is large where the observation impact is low (or non existant). We will see these same effects in many DA experiments.

2.4.4 Running EOLDAS from the command line

An alternative to writing your own python code for the front end is to use eoldas.py, which can be direcly run from the command line.

Help on command line options is available by typing::

```
eoldas_run.py --help
```

As an aid to setting up the correct python variables etc, a front end script, eoldas can also be accessed (in the bin directory).

```
eoldas_run.py --conf=confs/Identity.conf
```

N.B. Make sure the `eoldas_run.py` script is in your path. If you install the python packages for a single user, in UNIX it will usually be under `~/.local/bin/`. You may want to add that path to your users' path.

Application to MODIS reflectance data

We can now apply the concepts demonstrated above to real EO data from the MODIS sensors.

This is actually quite trivial to achieve as it involves the same basic configuration file as previously. This time, we will access it from `eoldas_run.py`.

2.4.5 The MODIS data file format

An example data file then is `data/modis_botswana.dat` which is MODIS reflectance data for a site in Botswana. The format of this file is 'BRDF', which looks like:

```
BRDF 67 7 645 858.5 469 555 1240 1640 2130  0.003 0.004 0.004 0.015 0.013 0.01 0.006
181 1 62.830002 -83.040001 37.910000 23.219999 0.032900 0.068600 0.018000 0.028400 0.098100 0.097
182 1 33.980000 99.540001 44.970001 39.290001 0.050500 0.091800 0.030000 0.045300 0.123200 0.1371
184 1 51.660000 99.300003 46.889999 42.349998 0.056200 0.106000 0.029500 0.053800 0.151600 0.1546
185 1 32.880001 -81.900002 40.540001 31.510000 0.039200 0.069600 0.022900 0.034400 0.089800 0.102
186 1 63.049999 100.470001 48.910000 45.169998 0.058200 0.138600 0.038900 0.055200 0.184600 0.183
187 1 6.430000 -79.459999 42.090000 35.259998 0.043700 0.074600 0.026400 0.038700 0.103100 0.1184
189 1 22.389999 97.720001 43.790001 38.750000 0.044500 0.088400 0.027100 0.040200 0.129500 0.1358
190 1 57.400002 -82.720001 38.029999 26.480000 0.050700 0.087700 0.024600 0.045200 0.119900 0.118
191 1 44.040001 99.419998 45.619999 41.990002 0.062800 0.107800 0.035600 0.057100 0.141600 0.1548
192 1 42.689999 -82.820000 39.290001 30.719999 0.041300 0.078800 0.022700 0.036600 0.106400 0.109
193 1 58.139999 99.800003 47.560001 45.000000 0.063500 0.124200 0.034900 0.060800 0.169700 0.1709
194 1 20.129999 -82.099998 40.730000 34.689999 0.038600 0.073200 0.024000 0.035100 0.106900 0.111
196 1 8.940000 96.330002 42.330002 38.389999 0.052200 0.090000 0.030300 0.047200 0.118400 0.13170
197 1 62.480000 -83.120003 36.610001 25.559999 0.053800 0.092700 0.025000 0.049200 0.118100 0.120
198 1 34.730000 99.230003 44.060001 41.840000 0.054500 0.103800 0.030300 0.051400 0.144800 0.1472
199 1 50.549999 -82.849998 37.750000 30.030001 0.044100 0.082200 0.025500 0.041300 0.104800 0.114
200 1 52.070000 100.010002 45.910000 45.029999 0.074300 0.132000 0.037400 0.069800 0.174200 0.169
201 1 32.130001 -82.889999 39.080002 34.240002 0.041500 0.077000 0.026200 0.038900 0.071200 0.108
202 1 63.400002 100.080002 47.880001 48.009998 0.082600 0.155400 0.043600 0.073800 0.191000 0.187
203 1 5.270000 -76.970001 40.590000 38.189999 0.053000 0.092500 0.026800 0.047500 0.122900 0.1214
205 1 23.219999 97.779999 42.220001 41.840000 0.067000 0.113900 0.034600 0.060000 0.143100 0.1485
206 1 56.959999 -83.139999 35.910000 29.340000 0.065800 0.102000 0.033800 0.062000 0.117800 0.113
```

The first line is a header, as with the previous 'PARAMETERS' format we saw. The first word on the header must be `BRDF` (a # can be included for compatitbilty with some other formats).

The second item on the header line is the number of samples in the file (92 here). The third item is the number of wavebands represented (7 here) then this is followed by wavelengths, wavebands, or waveband identifiers.

If a single float number is given, it is assumed to represent a narrow waveband centred around that wavelength. Wavelength is assumed to be in nm. If it is two float numbers connected by a dash (e.g. 45-550) then it is taken to be a tophat function representation of a waveband, where the two number represent the minimum and maximum

wavelength respectively. If some other code is found (e.g. `B2`), it is assumed to be a tag associated with a particular waveband. In this case, any spectral libraries loaded in the configuration files are searched to see if they contain a definition of this waveband. If nothing is found and it cannot be interpreted as a number, it is assigned an arbitrary wavelength (index starting from 0) and should not be interpreted as a wavelength identifier.

The 7 columns following the wavebands specify the standard deviation in the reflectance data. All subsequent lines contain reflectance, as a function of 'locational', 'control' and spectral information. The first column here defines location (time), columns 2 to 6 are 'control' variables: things the observation waries as a function of that are not spectral or locational. In this case, they are (all angles are in degrees):

a data mask (1 for good), # view zenith angle, # view azimuth angle, # solar zenith angle, # solar azimuth angle.

The final 7 columns give the reflectance in the 7 wavebands for each location defined.

The data span days of year 181 to 272 inclusive. There is a lot of day to day variation in the data, but a clear underlying trend in the Near infrared. We will set ourselves the task of trying to use eoldas to filter the dataset so that the trend becomes more apparent. We will also try to extrapolate from the sample days to the whole year (which is clearly quite a challenge, but instructive for understanding uncertainty).

We will apply the same data assimilation components as previously, i.e. an Identity operator for the NIR reflectance (in other words, we take the reflectance as a prior estimate of what we wish to estimate) and a regularisation filter implemented as a constraint on first order derivatives at lag one day. We will suppose that we know the uncertainty in this model (the temporal constraint) to be expressed as a standard deviation of .002 (1/500), i.e. the root mean squared deviation from one day to the next is expected to be around .002. An examination of the dataset will show that the NIR reflectance roughly increases from around 0.1 to 0.2 over about 100 days, so we could use a gamma value of 1000. However, we don't want to impose that constraint too strongly, so we will instead use a gamma of 500. We are not trying to fit a trendline here, just to smooth the dataset so that we can detect the underlying behaviour. The result should however not be very sensitive to this guess and can always be explored using approaches such as cross validation.

2.4.6 The configuration file

We have a configuration file set up in `config_files/Identity2.conf`. This provides a full description of the problem we wish to solve, which is to reduce noise in the NIR observation series.

```
# An EOLDAS configuration file for
# a simple Identity operator and a regulariser

[parameter]
location = ['time']
limits = [[1,365,1]]
help_limits="define the limits for the required locational information"
names = "gamma_time 858.5".split()
solve = 0,1
datatypes = x

[parameter.result]
filename = output/Identity/MODIS_botswana.params
format = 'PARAMETERS'
help_filename='Set the output state filename'

[parameter.x]
datatype = x
names = $parameter.names
default = 500,0.1
state= data/modis_botswana.dat
help_default="Set the default values of the states"
apply_grid = True
sd = [1.]*len($parameter.names)
bounds =  [[0.000001,1000000],[0,1]]

[general]
```

```
doplot=True
help_do_plot='do plotting'
calc_posterior_unc=False

[operator]
modelt.name=DModel_Operator
modelt.datatypes = x
obs.name=Operator
obs.datatypes = x,y

[operator.modelt.x]
names = $parameter.names
sd = [1.0]*len($operator.modelt.x.names)
datatype = x

[operator.modelt.rt_model]
model_order=1
help_model_order='The differential model order'
wraparound=periodic,365
#wraparound=None

[operator.obs.x]
names = $parameter.names[1:]
sd = [1.0]*len($operator.obs.x.names)
help_sd='Set the observation sd'
datatype = x

[operator.obs.y]
control = [mask,vza,vaa,sza,saa]
names = $parameter.names[1:]
sd = [0.015]*len($operator.obs.x.names)
#help_sd="set the sd for the observations"
datatype = y
state = data/modis_botswana.dat
help_state = "Set the y state vector"

[operator.obs.y.result]
filename = output/Identity/Botswana_fwd.params
format = 'PARAMETERS'
help_filename = 'Set the fwd modelling result filename'
```

In this, we can see the required location field defined in the `[parameters]` section (i.e. `parameters.location=[[1,365,1]]`). The control variables are associated with the observation operator, here given as `[mask,vza,vaa,sza,saa]`.

To run eoldas with this configuration then all we need do is type:

```
eoldas_run.py --conf=config_files/eoldas_config.conf \
  --conf=config_files/Identity2.conf --calc_posterior_unc
```

This writes the files:

`output/Identity/MODIS_botswana.params`: state estimation (smoothed reflectance)

```
 181.000000    1.000000  500.000000    0.095014    0.000000    0.005513
 182.000000    1.000000  500.000000    0.095247    0.000000    0.005255
 183.000000    1.000000  500.000000    0.095541    0.000000    0.005063
 184.000000    1.000000  500.000000    0.095835    0.000000    0.004840
 185.000000    1.000000  500.000000    0.095949    0.000000    0.004669
 186.000000    1.000000  500.000000    0.096530    0.000000    0.004541
 187.000000    1.000000  500.000000    0.096364    0.000000    0.004451
 188.000000    1.000000  500.000000    0.096585    0.000000    0.004396
 189.000000    1.000000  500.000000    0.096806    0.000000    0.004293
 190.000000    1.000000  500.000000    0.097176    0.000000    0.004216
```

```
 191.000000    1.000000  500.000000    0.097714    0.000000    0.004161
```

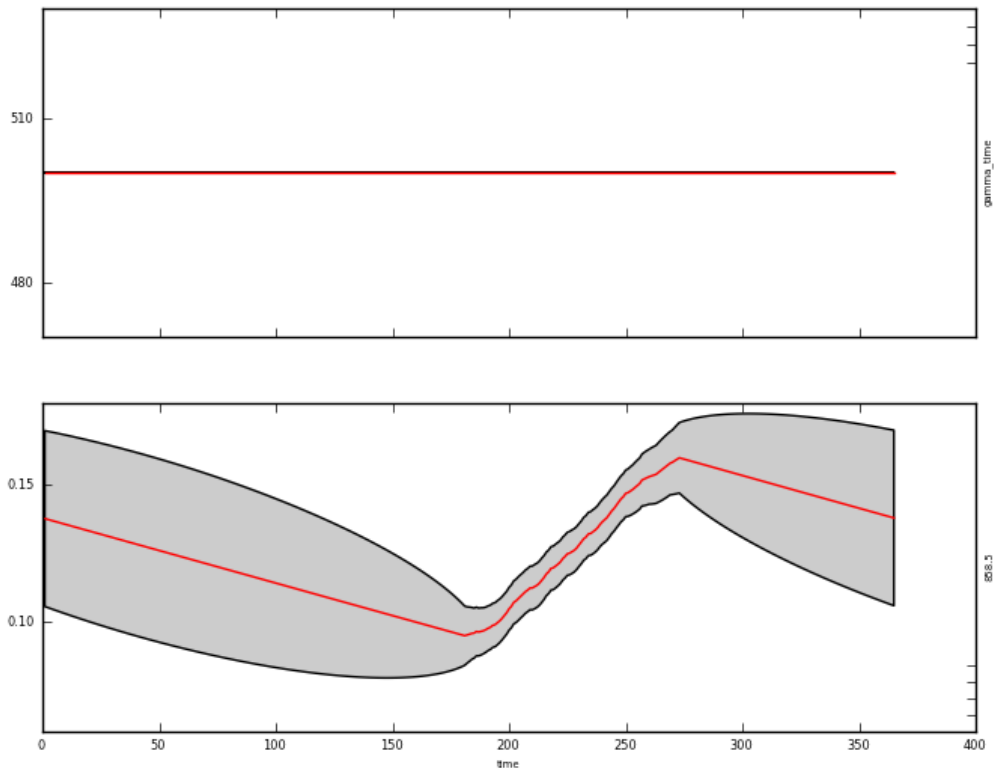
`output/Identity/Botswana_fwd.params`: forward modelling of y

```
#PARAMETERS time mask,vza,vaa,sza,saa 858.5 sd-858.5
 181.000000    0.000000    0.095014    0.000000
 182.000000    0.000000    0.095247    0.000000
 184.000000    0.000000    0.095835    0.000000
 185.000000    0.000000    0.095949    0.000000
 186.000000    0.000000    0.096530    0.000000
 187.000000    0.000000    0.096364    0.000000
 189.000000    0.000000    0.096806    0.000000
 190.000000    0.000000    0.097176    0.000000
 191.000000    0.000000    0.097714    0.000000
```

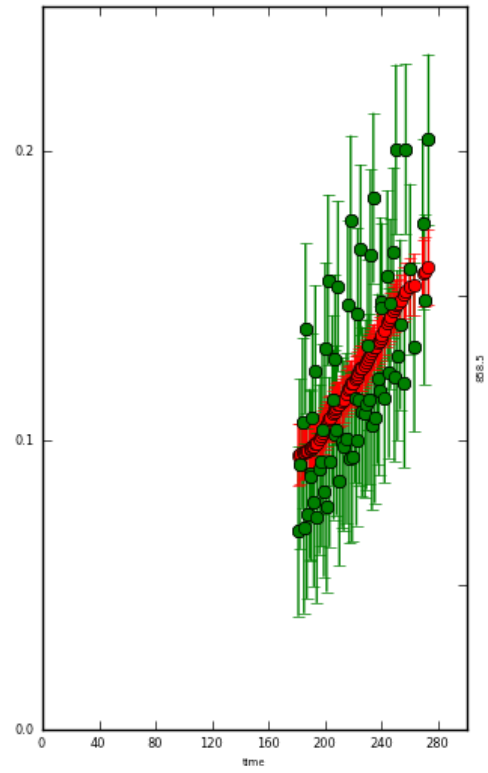
as well as the orignal data in `output/Identity/Botswana_fwd.params_orig`

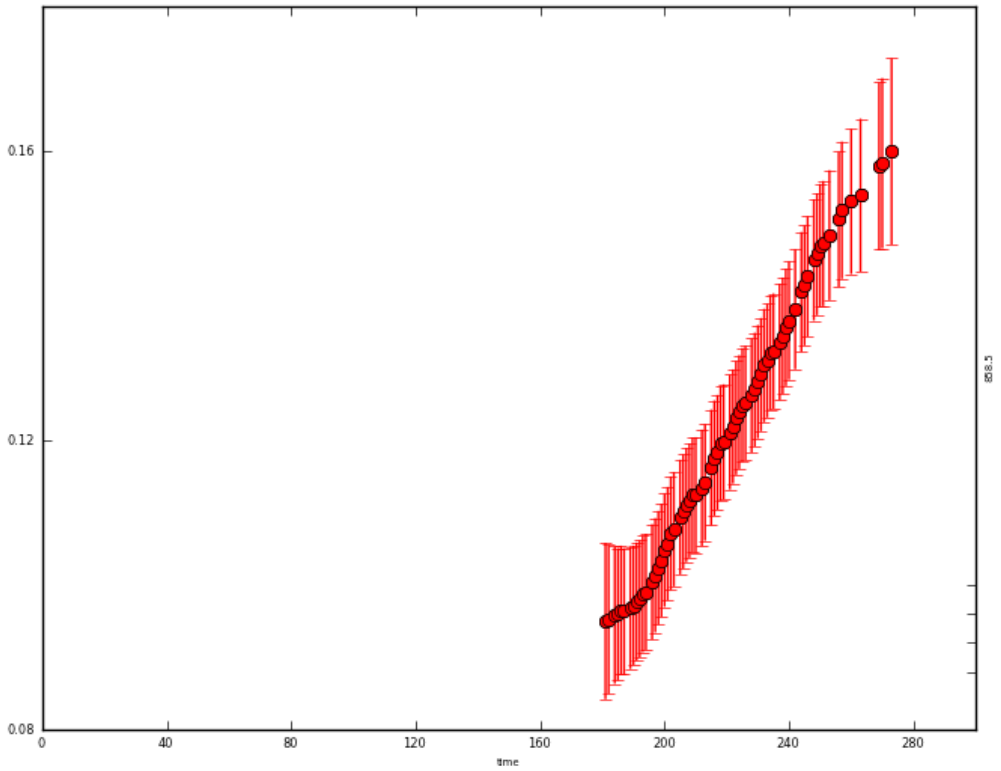
```
#PARAMETERS time mask,vza,vaa,sza,saa 858.5 sd-858.5
 181.000000    0.000000    0.068600    0.015000
 182.000000    0.000000    0.091800    0.015000
 184.000000    0.000000    0.106000    0.015000
 185.000000    0.000000    0.069600    0.015000
 186.000000    0.000000    0.138600    0.015000
 187.000000    0.000000    0.074600    0.015000
 189.000000    0.000000    0.088400    0.015000
 190.000000    0.000000    0.087700    0.015000
 191.000000    0.000000    0.107800    0.015000
```

with appropriate graphics for the state:



and the y data:





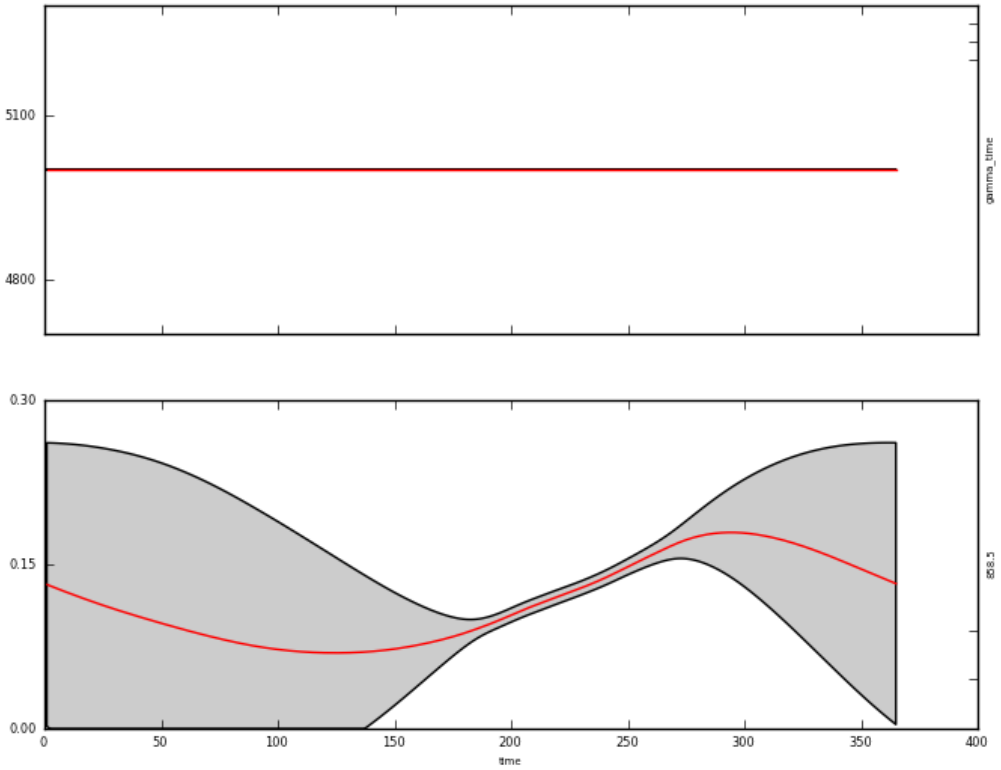
The resultant state data are quite instructive: where we have observations, the uncertainty is reduced from 0.015

to around 0.004 (the actual degree of noise reduction depends on the value of gamma used). Where there are no data, the uncertainty grows to around 0.017. It is slightly reduced at the year start/end because of the wrparound condition used here.

In this case, we have used a first order differential constraint with a periodic boundary condition. These are quite important in this case: we only have observations in a limited time window, so using a periodic boundary condition is one way to place some form of constraint at what happens when we have no data. The first order differential model will in essence perform a linear interpolation where there are no data, which is probably appropriate for this case. You can try changing the model order to see what happens. Run e.g.:

```
eoldas_run.py --conf=config_files/eoldas_config.conf \
  --conf=config_files/Identity2.conf --calc_posterior_unc \
  --operator.modelt.rt_model.model_order=2 \
  --parameter.x.default=5000,0.1 \
  --operator.obs.y.result.filename=output/Identity/Botswana_fwd.params2 \
  --parameter.result.filename=output/Identity/MODIS_botswana.params2
```

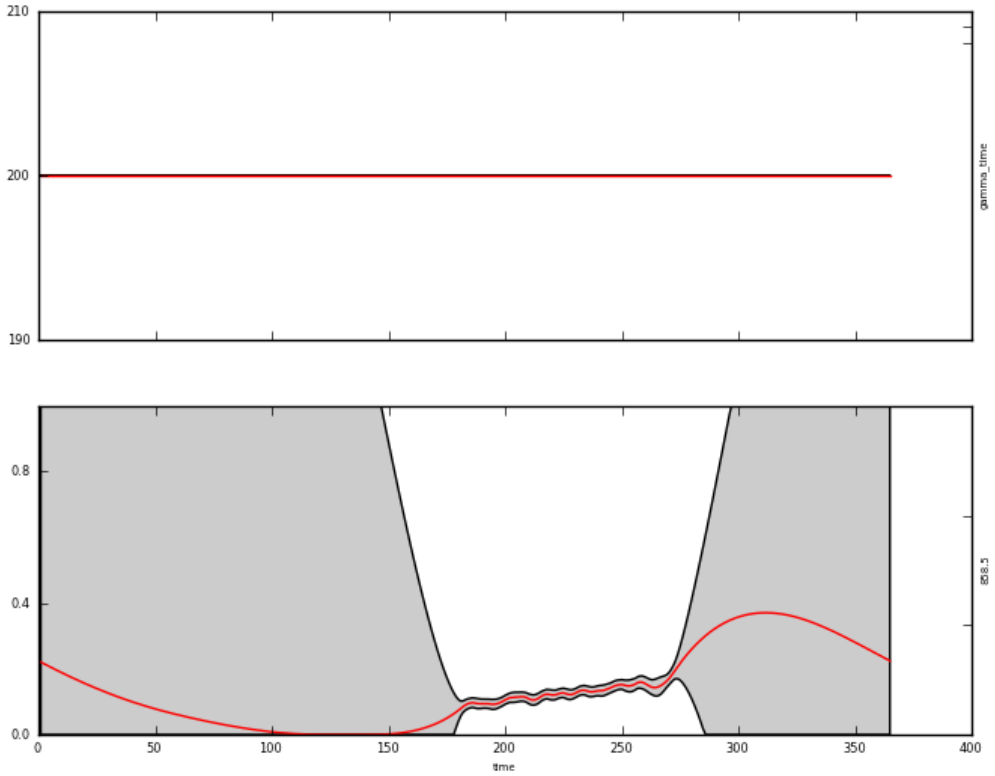
and have a look at `output/Identity/MODIS_botswana.params2.plot.x.png`:



The assumed behaviour is quite different to the first order differential constraint outside of the observations. With a high value of gamma, the result is essentially a straight line where there are observations. The influence of the wraparound condition is also clear here.

If we used a lower gamma, we would see some features of using a second order model:

```
eoldas_run.py --conf=config_files/eoldas_config.conf \
  --conf=config_files/Identity2.conf --calc_posterior_unc \
  --operator.modelt.rt_model.model_order=2 \
  --parameter.x.default=200,0.1  \
  --operator.obs.y.result.filename=output/Identity/Botswana_fwd.params3  \
  --parameter.result.filename=output/Identity/MODIS_botswana.params3
```



Where there are observations, the second order difference constraint would have high frequency oscillations. This can be a positive feature or an annoyance: it all depends on how you expect the function to behave. It is not an arbitrary choice: the user has imposed a particular expectation of behaviour here through the model. There clearly needs to be some evidence for choosing one form of model over another, although that is not always straightforward.

One other interesting feature of this result is that the mean estimate is bounded (0.0,1.0) which is reasonable for reflectance data (theoretically, BRF can go above 1, but this is rarely observed). This means that the mean value (the red line) is forced to stay above 0 (days 100 to 150) even though the apparent trend from the observations might otherwise make it go below zero. This condition is imposed in the configuration by the line:

```
bounds =  [[0.000001,1000000],[0,1]]
```

One further thing to note about this result is that when the uncertainty is so large (as is the case here when we are extrapolating) that the confidence interval spans more than the entire data range (0,1) the mean and standard deviation described in the Gaussian statistics are a little meaningless or at least should be interpreted with caution. It might in this case be better to describe the data (in the extrapolated region) as simply being somewhere between the physical limits.

A final comment is that actually, a large amount of the departure of the signal from our smooth trajectory, the high frequency variation that we have treated as noise here, can most likely be described by considering the physics os scattering by the surface (there are, on the whole, BRDF effects). We will return to this later.

As well as raising a few issues with regard to model selection, we have demonstrated further use of the eoldas command line for quickly changing some terms in an experiment. However, it is all very well in showing that eoldas can ingest satellite data and apply constraints to provide an estimate of state variables, but the state variables here (reflectance) do not directly help us monitor the properties of the land surface or link to process models.

To do that using EO data, we generally need more complex obeseravtion operators. These are dealt with in the next section.

2.5 Radiative transfer modelling for optical remote sensing

2.5.1 Radiative transfer modelling for optical remote sensing

Introduction

Our main focus will be on using the EOLDAS data assimilation techniques to understand the land surface using optical remote sensing data. Optical sensors capture reflected solar radiation in the 400 to 2500nm range, over a number of wavebands. The signal on these bands is a combination of processes that influence the scattering of photons, such as interactions of photons within vegetation canopies, interactions with aerosols in the atmosphere, gaseous absorption processes, etc. Since all these processes contribute to the signal, it is important to have a physical understanding to separate their contributions and produce accurate estimates of the combined state of the atmosphere and land surface.

Typically, RT models of the land surface model the interactions of photons and plant components (leaves, trunks, etc.) above a surface layer (usually soil, but also snow). Atmospheric radiative transfer models are needed to account for water vapour and ozone absorption in the atmosphere, as well as for the effect of aerosols. One can think of this arrangement as a stacking of models, going from bottom to top as: soil model, vegetation model and atmospheric model.

Our main focus is the study of the land surface using DA techniques. The proposed RT models predict, for a given wavelength and acquisition geometry, directional reflectance of the land surface. The parameters that govern these models are typically to do with soil structure (soil brightness and roughness terms) and the vegetation architecture (leaf area index (LAI), leaf angle distribution, ...), as well as leaf biochemical parameters (chlorphyll a+b concentration, dry matter, leaf structural parameters...). These paraeters are the ones that form our land surface state vector, and they will vary with time and location, but provide a full description of the surface.

We note that these models are highly non-linear, and combined with only modest observations available in terms of angular and spectral sampling, as well as the contribution of thermal noise in the acquisitions, result in an*ill-posed problem*: typically, the available observations do not provide enough constrains to the models, and this results in state vector elements being poorly determined, equivalently having large uncertainties. We shall see that these situation can be improved by the use of data assimilation techniques.

We will first consider a simple RT model for continuous canopies under a soil layer, with a leaf optical model embedded. The soil layer is Lambertian, and it is assumed that its brightness at a specific waveband is a linear combination of a number of spectral basis functions. By default, these are the basis functions defined in Price 1991 , but they can be easily changed by users. Note that the model will oversimplify the directional effects of typical rough land surfaces, and that the Price spectral basis functions really only apply to soils at field capacity. Also, the effect of e.g. snow is not considered.

The leaf model is the widely used PROSPECT model, which treats leaves as a stack of thin plates. Other parameters control the absorption of energy at different wavelengths. The parameters of the PROSPECT model are

1. N, the number of leaf layers,
2. C_{ab}, the concentration of chlorophyll a+b $[\mu g cm^{-2}]$,
3. C_{car}, the concentration of carotenoids $[\mu g cm^{-2}]$,
4. C_w, equivalent leaf water $[cm]$,
5. C_{dm}, dry matter $[\mu g cm^{-2}]$,
6. C_{sen}, the proportion of senescent material (fractional, between 0 and 1).

The canopy model chosen is Semidiscrete by Gobron et al. 1997, for which an adjoint has been develoepd here to allow more rapid state estimation. This model assumes a continuous canopy and is governed by LAI, leaf area distribution (discretised into five different classes), two terms that control the so-called "hotspot" effect, and three spectral terms (one for the soil and two for leaves), which are fed in from PROSPECT and the spectral soil model. In total, the state vector has thirteen or fourteen components (four related to the soil, six or seven to the leaf properties and three to the canopy). This allows the simulation of the directional reflectance for narrow spectral wavebands and for a specified illumination/acquisition geometry.

2.5.2 Solving for a single observation

We first apply the eoldas to the problem of attempting a state estimate from a single satellite observation. These data have multiple wavebands, but we consider here only a single location and date.

An example data file for this is some MERIS data over a field site in Germany. The file is `MERIS_WW_1_A_1.brf`, the start of which looks like:

```
#BRDF 10 15 412 442 489 509 559 619 664 680 708 753 761 778 864 884 900 0.01 0.01 0.01 0.01 0.01 0
108 1 4.307049 286.659393 42.972881 152.909012  0.019400 0.025100 0.034100 0.035800 0.059700 0.043
114 1 19.283617 288.855164 40.170044 156.520325  0.022000 0.027400 0.036100 0.038300 0.058600 0.04
155 1 32.146313 291.063110 30.257147 156.645920  0.010100 0.014900 0.022500 0.027900 0.054500 0.03
174 1 34.970631 291.616272 29.384569 155.618637  0.016800 0.020800 0.027200 0.031700 0.064400 0.04
178 1 4.285298 286.663055 31.555040 144.463577  0.024900 0.028300 0.033600 0.037000 0.062700 0.042
190 1 32.115547 291.067474 30.814306 153.679443  0.016700 0.023600 0.034600 0.042000 0.072200 0.06
190 1 32.115547 291.067566 30.814367 153.679428  0.014000 0.021200 0.032700 0.040300 0.071000 0.06
197 1 8.136833 287.214844 33.390018 145.816544  0.041900 0.049500 0.061600 0.069900 0.099200 0.113
232 1 8.137174 287.211914 41.582573 151.810349  0.033900 0.042800 0.056700 0.062800 0.092400 0.090
264 1 0.380700 206.624115 52.586159 157.652710  0.065000 0.073300 0.086500 0.089100 0.101900 0.113
```

We build a suitable configuration file `config_files/meris_single.conf`

```
# configuration file
# note that some words are protected: values, keys
# so dont use them
# other than that, you can set any level of hierarchy in the
# representation
# this next part serves no purpose other than
# to show how different options can be set

# It is a requirement that parameter.names be defined

[parameter]
location = ['time']
limits = [[1,365,1]]
names=gamma_time,xlai, xhc,  rpl,  xkab, scen, xkw, xkm,   xleafn, xs1,xs2,xs3,xs4,lad
solve = [1]*len($parameter.names)
help_solve='flags for which state vector elements to solve for'

[parameter.result]
filename = 'output/meris/MERIS_WW_1_A_1.params'
help_filename="state vector results file"
format = 'PARAMETERS'

[parameter.assoc_solve]
xkw = 0
gamma_time = 0
lad = 0
xs3 = 0
xs4 = 0
xhc = 0
rpl = 0

[parameter.x]
datatype = x
names = $parameter.names
default = [0.05]*len($parameter.names)
help_default = "Set the parameter default values"
apply_grid = True
sd = [1.]*len($parameter.names)
bounds = [[0.01,0.99]]*len($parameter.names)
####state = fullState.dat
invtransform=$parameter.names
transform=$parameter.names
```

```
[parameter.x.assoc_transform]
xlai=np.exp(-xlai/2.)
xkab=np.exp(-xkab/100.)
xkw=np.exp(-xkw*50.)
xkm=np.exp(-100.*xkm)

[parameter.x.assoc_invtransform]
xlai=-2.*np.log(xlai)
xkab=-100.*np.log(xkab)
xkw=-(1./50.)*np.log(xkw)
xkm=-(1./100.)*np.log(xkm)

[parameter.x.assoc_bounds]
gamma_time = 0.000001,100000
xlai = 0.05,0.99
xhc = 0.05,10.0
rpl = 0.01,0.10
xkab = 0.1,0.99
scen = 0.001,1.0
xkw = 0.01,0.99
xkm = 0.3,0.9
xleafn = 0.9,2.5
xs1 = 0.01, 4.
xs2 = 0.01, 5.
xs3 = 0., 0.
xs4 = 0.,0.
lad = 1,5


[parameter.x.assoc_default]
####gamma_time = 0.01
xlai = 0.95
xhc = 5
rpl = 0.01
xkab = 0.95
scen = 0.001
xkw = 0.95
xkm = 0.35
xleafn = 1.5
xs1 = 1.0
xs2 = 0.001
xs3 = 0
xs4 = 0
lad = 5
help_lad = 'lad value'

[general]
is_spectral = True
calc_posterior_unc=False
write_results=True
doplot=True
plotmod=20
plotmovie=True

[general.optimisation]
randomise=False

[operator]
obs.name=Observation_Operator
obs.datatypes = x,y

[operator.obs.rt_model]
```

```
model=semidiscrete1
use_median=True
help_use_median = "Flag to state whether full bandpass function should be used or not.\nIf True, t
bounds = [400,2500,1]
help_bounds = "The spectral bounds (min,max,step) for the operator'
ignore_derivative=False
help_ignore_derivative = "Set to True to override loading any defined derivative functions in the

[operator.obs.x]
names = $parameter.names[1:]
sd = [1.0]*len($operator.obs.x.names)
datatype = x

[operator.obs.y]
control = 'mask vza vaa sza saa'.split()
names = "412 442 489 509 559 619 664 680 708 753 761 778 864 884 900".split()
sd = "0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01".split()
datatype = y
state = data/meris/MERIS_WW_1_A_1.brf
help_state='set the obs state file'

[operator.obs.y.result]
filename = 'output/meris/MERIS_WW_1_A_1.fwd'
help_filename = 'forward modelling results file'
format = 'PARAMETERS'
```

You can note several things about the semidiscrete observation oeprator from this:

```
names=gamma_time,xlai, xhc,  rpl,  xkab, scen, xkw, xkm,   xleafn, xs1,xs2,xs3,xs4,lad
```

The (apparent) state variables, other than gamma_time that is (potentially) used for the differential operator are:

1. xlai : LAI, the single sided leaf area per unit ground area
2. xhc : The canopy height
3. rpl : The leaf radius/dimension (same units as canopy height)
4. xkab : C_{ab}, the concentration of chlorophyll a+b $[\mu g cm^{-2}]$,
5. scen : C_{sen}, the proportion of senescent material (fractional, between 0 and 1).
6. xkw : C_w, equivalent leaf water $[cm]$,
7. xkm : C_{dm}, dry matter $[\mu g cm^{-2}]$,
8. xleafn : N, the number of leaf layers,
9. xs1 : Soil PC1 (soil brightness)
10. xs2 : Soil wetness
11. xs3 : not used
12. xs4 : not used
13. lad : leaf angle distribution (coded 0-5). See Semidiscrete code for details.

However, the section `[parameter.x.assoc_transform]` contains:

```
####state = fullState.dat
invtransform=$parameter.names
```

which sets default state transformations and inverse transforms to be identity (e.g. xlai = xlai).

Then the section:

```
[parameter.x.assoc_transform]
xlai=np.exp(-xlai/2.)
xkab=np.exp(-xkab/100.)
xkw=np.exp(-xkw*50.)
```

overrides this for the specifc states mentioned to tell the eoldas that there is a transformation (in the RT model code) between true state (e.g. LAI) and the state we solve for. The reason for using a transformation is in an attempt to approximately linearise the sensitivity of $H(x)$ to the state variable. This is a good idea generally, for avoiding getting trapped in local minima when optimising and also because Gaussian statistics are more appropriate for the linearised state. The inverse transforms are covered in the sections `[parameter.x.assoc_invtransform]`

```
[parameter.x.assoc_invtransform]
xlai=-2.*np.log(xlai)
xkab=-100.*np.log(xkab)
xkw=-(1./50.)*np.log(xkw)
```

The state vectors will be saved to file in their transformed form (as this is appropriate for the stored uncertainty information), but plots are generated for the inverse transformed states. The user has access to these inverse transforms in the code as x.transform and x.invtransform.

Second, we note that some of the state variables are 'switched off' for optimisation in the section `parameter.solve`. Initially, all solve states are set to 1:

```
solve = [1]*len($parameter.names)
```

Then we switch selected states to 0:

```
[parameter.assoc_solve]
xkw = 0
gamma_time = 0
lad = 0
xs3 = 0
xs4 = 0
xhc = 0
```

In the `[general]` section, we have set the flags:

```
write_results=True
doplot=True
plotmod=20
```

which tell eoldas to generate plots, and to keep updating them every 'plotmod' calls to the operator. This allows the user to visualise how the data assimilation is proceeding, by viewing the appropriate png files. The file names are formed from any results.filename fields found in a state.

Finally, we note the section `[operator.obs.rt_model]`. This tells the eoldas parameters specific to the operator being used. In the case of an observation operator, this must include `operator.obs.rt_model.model`. Here, we have:

```
[operator.obs.rt_model]
model=semidiscrete1
use_median=True
help_use_median = "Flag to state whether full bandpass function should be used or not.\nIf True, t
bounds = [400,2500,1]
help_bounds = "The spectral bounds (min,max,step) for the operator'
ignore_derivative=False
```

where we set the model to `semidiscrete1`, which will have been installed with the eoldas package (it is a dependency)

We can run this configuration with:

```
~/.local/bin/eoldas_run.py --conf=config_files/eoldas_config.conf \
  --conf=config_files/meris_single.conf \
  --parameter.limits='[[232,232,1]]' --calc_posterior_unc
```

which tells eoldas to solve for 8 state varibles: `xlai, xkab, scen, xkw, xkm, xleafn, xs1, xs2` from any meris data found on day 232.

Here, since we are using an observation on a single data, we do not include the dynamic model, and merely attempt to solve with the observation operator. This process, with a single operator (the radiative transfer model), can also be called 'model inversion' where we attempt to estimate the state vector elements known (and expressed in the radiative transfer model) to control the spectral bidirectional reflectance of vegetation canopies). In truth we cannot claim that this is the optimal solution based only on the model and observations, as we also have the implicit constraint of bounds to the state estimate. This is a form of prior information that we use to help constrain the problem. Using an Identity Operator as we have done in previous exercises is, in many ways just another way of expressing this same idea, just with a different statistical distribution. With 'hard' bounds to the state vector elements we are assuming a 'flat' probability of occurrence between those bounds. If we use an Identity Operator and describe y as a Gaussian distribution, we have a 'soft' bound to the problem. In many cases there are physical limits to states, for example, xlai must lie in the bound [0,1] or more practically away from the edges of those bounds, (0,1). Similarly, plant height cannot physically be negative.

Results are written to `output/meris/MERIS_WW_1_A_1.params`:

```
#PARAMETERS time gamma_time xlai xhc rpl xkab scen xkw xkm xleafn xs1 xs2 xs3 xs4 lad sd-gamma_tir
 232.000000    0.050000    0.721891    5.000000    0.010000    0.780271    0.001000    0.950000
```

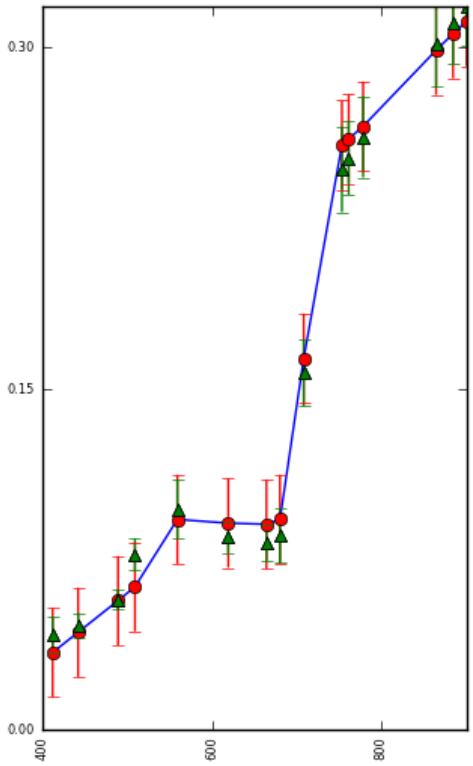
with the foward modelled results in `output/meris/MERIS_WW_1_A_1.fwd`:

```
#PARAMETERS time mask vza vaa sza saa 412 442 489 509 559 619 664 680 708 753 761 778 864 884 900
 232.000000    1.000000    8.137174  287.211914   41.582573  151.810349    0.041480    0.045971
```

and the original data in `output/meris/MERIS_WW_1_A_1.fwd_orig`:

```
#PARAMETERS time mask vza vaa sza saa 412 442 489 509 559 619 664 680 708 753 761 778 864 884 900
 232.000000    1.000000    8.137174  287.211914   41.582573  151.810349    0.033900    0.042800
```

with graphics:



which is a plot of MERIS reflectance as a function of wavelength. The red dots show the orginal data points with assumed uncertainty shown as 95% confidence intervals (0.01 * 1.96 here). The green line is the modelled version, with the error bars indicating the 95% confidence intervals for each waveband (slightly, but not greatly lower than the prior uncertainties).

This shows that the eoldas is able to replicate the observed reflectance very well. If we can do this, then why do we need data assimilation? The answer to that comes from looking back at the solved state information.

For `xlai` (i.e. $\exp(-LAI/2)$) we retrieve a value of 0.721891, i.e. LAI of 0.652. This might well be plausible for the particular location, but we see that we have an uncertainty of 0.654474 on this. Since xlai is bounded [0,1] this means that this is a very unreliable estimate.

For some of the model states that we requested to be solved we do even worse, `xkw` (also bounded [0,1]) has an apparent uncertainty of 46.326810, though the reason for this is simply that the MERIS instrument does not sample at wavelengths at which reflectance is very sensitive to leaf water content. Interestingly, one of the lower uncertainties is 0.166161 for `xkab`. This is because MERIS *does* sample at several wavebands that are sensitive to chlorophyll content. The retrieved value here is 0.780574 which corresponds to a concentration of 24.8 $[\mu g cm^{-2}]$.

So, not only is our knowledge of these state variables we require actually quite poor from a single observation, but also, if we were to look at the full inverse Hessian, we would see high degrees of correlation between the state estimates.

This is not surprising: we are trying here to retrieve 8 states from measurements in 15 wavebands. We have assumed here an uncertainty of 0.01 in the surface reflectance, which could be a little high for some bands, but note that we have assumed the error independent for each wavelength, which it most certainly would not be (in that case, and in the absence of any further information on the error correlation structure, it may be appropriate to inflate the standard deviation).

On the positive side, the optimisation did at least converge to a viable solution, and gives a final value of J of around 3.83. Remember that J is essentially half the sum of the squared differences between modelled and measured reflectance, over 15 wavebands here, gives an MSE equivalent of 0.51 and a RMSE of 0.71, which we can loosely interpret as meaning that the 'average' departure between what we measure and what we simulate is

around 0.71 standard deviations.

2.5.3 Simulating surface reflectance with the EOLDAS model operator

Even though the state estimate is a little poor, we can take it and simulate what we would see with other sensors (i.e. simulate different sensors). This demonstrates how the state vector, phrased as biophysical variables, allows us to translate information from one sensor to another (and thus allows us to combine information from multiple sensors as we shall see).

We build a configuration file for the MODIS instrument as `config_files/modis_single.conf`.

This is mainly the same as the meris configuration file other that a few items refering to the input state (which is the output of the meris run here: `output/meris/MERIS_WW_1_A_1.params`), the band names and uncertainty, and the input observations file, `data/brdf_WW_1_A_1.kernelFiltered.dat`.

```
# configuration file
# a minimal configuration file
# with overrides for meris_single.conf

[parameter.result]
filename = 'output/modis/MODIS_WW_1_A_1.params'

[parameter.x]
state = 'output/meris/MERIS_WW_1_A_1.params'

[operator.obs.y]
names = "465.6 553.6 645.5 856.5 1241.6 1629.1 2114.1".split()
sd = "0.003 0.004 0.004 0.015 0.013 0.01 0.006".split()
state = data/brdf_WW_1_A_1.kernelFiltered.dat

[operator.obs.y.result]
filename = 'output/modis/MODIS_WW_1_A_1.fwd'
```

We can, for instance then run this with:

```
eoldas_run.py --conf=config_files/eoldas_config.conf \
    --conf=config_files/meris_single.conf \
    --parameter.limits='[[232,232,1]]' \
    --passer --conf=config_files/modis_single.conf
```

We need to load the MERIS configuration file before the MODIS one so that the latter overrides and options in the former. There are some dangers to using a whole string of configuration files as it can be difficult to keep track of information, but the software allows you to define as many as you want (up to any system limits to command lines etc.).

You can confirm that the configuration file loading is as you would expect by looking at the log file `logs/modis_single.log`. You really should look at that the first time you generate a new configuration file or you might end up mis-interpreting results.

Alternative to writing a new configuration file, we could have overridden the meris configuration settings with command line options. That has some of the same issues as having too many configuration files though as it could become difficult to keep track of information.

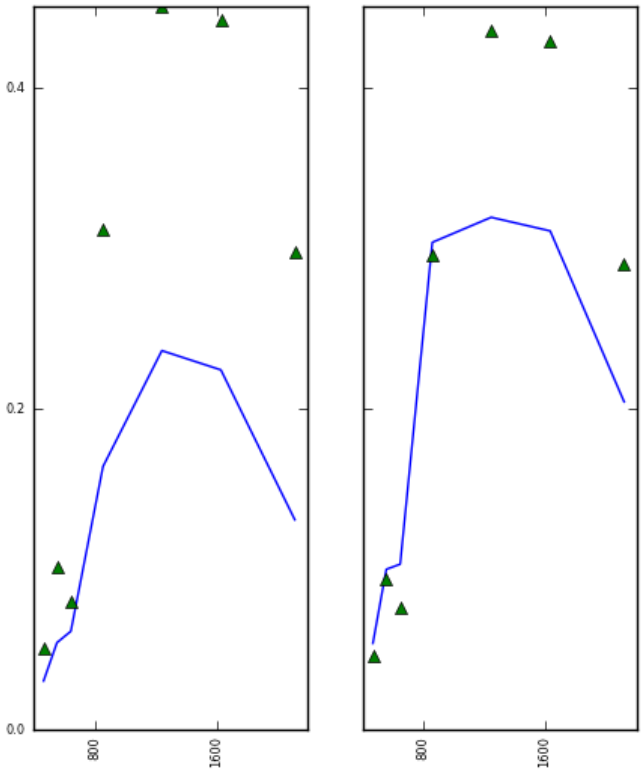
The flag `--passer` here tells eoldas *not* to perform parameter estimation, but just to do a 'forward' simulation from the state that we load.

The result we are interested in is written to `output/modis/MODIS_WW_1_A_1.fwd` and `output/modis/MODIS_WW_1_A_1.fwd_orig`:

```
#PARAMETERS time mask vza vaa sza saa 465.6 553.6 645.5 856.5 1241.6 1629.1 2114.1 sd-465.6 sd-553
 232.000000    1.000000   39.080000 -242.100000   38.990000    0.000000    0.050558    0.101616
 232.000000    1.000000   42.220000   51.240000   42.340000    0.000000    0.046082    0.094160
```

```
#PARAMETERS time mask vza vaa sza saa 465.6 553.6 645.5 856.5 1241.6 1629.1 2114.1 sd-465.6 sd-55
 232.000000    1.000000   39.080000 -242.100000   38.990000    0.000000    0.030300    0.054200
 232.000000    1.000000   42.220000   51.240000   42.340000    0.000000    0.053800    0.099900
```

with the plot in



The result is rather poor for both of the MODIS observations on day 232, especially at longer wavelengths. This could be due to factors such as poor atmospheric modelling of either the MODIS or MERIS datasets, but it could equally be just due to the large uncertainty in the state estimates obtained from the MERIS data alone.

So, the information content of the MERIS data alone for a single observation only poorly constrain the estimate of the state variables. If we had some prior expactation of those states, we could use that to help constrain the estimate and we will proceed onto that in the next section.

In the meantime, it is instructive to see how we can use *both* the MERIS and MODIS observations to constrain the state estimate for this single day dataset.

Before that, to get an estimate from MODIS alone we create a modified form of `config_files/modis_single.conf`, `config_files/modis_single_a.conf`:

```
# configuration file
# a minimal configuration file
# with overrides for meris_single.conf

[parameter.result]
filename = 'output/modis/MODIS_WW_1_A_1.params_a'

[operator.obs.y]
names = "465.6 553.6 645.5 856.5 1241.6 1629.1 2114.1".split()
sd = "0.003 0.004 0.004 0.015 0.013 0.01 0.006".split()
state = data/brdf_WW_1_A_1.kernelFiltered.dat

[operator.obs.y.result]
```

```
filename = 'output/modis/MODIS_WW_1_A_1.fwd_a'
```

We have changed the output filenames so as not to overwrite the previous results and remove the state definition for parameter.x (i.e. we don't use the information from MERIS in this example).

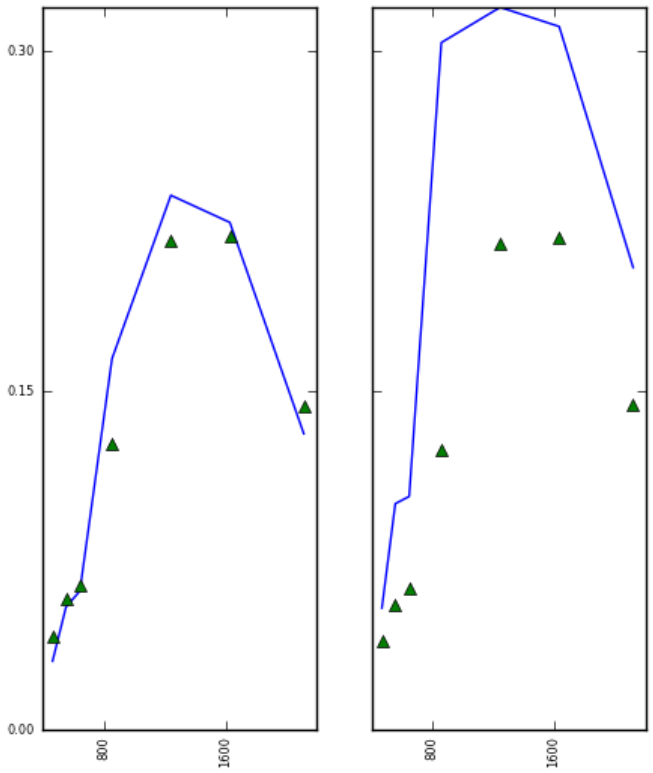
Similarly to before, we run:

```
eoldas_run.py --conf=config_files/eoldas_config.conf \
  --conf=config_files/meris_single.conf --passer \
  --parameter.limits='[[232,232,1]]'  \
  --conf=config_files/modis_single_a.conf
```

The state estimate result is in `output/modis/MODIS_WW_1_A_1.params_a`

```
#PARAMETERS time gamma_time xlai xhc rpl xkab scen xkw xkm xleafn xs1 xs2 xs3 xs4 lad sd-gamma_ti
 232.000000    0.050000    0.950000    5.000000    0.010000    0.950000    0.001000    0.950000
```

with the graphic in



This result is not particularly good, and if we examine the s of the optimisation we will see a warning: ABNORMAL_TERMINATION_IN_LNSRCH. This is simly not a viable result, even though we have wider spectral sampling than for MERIS. That may be partly down to particular characteristoics of these samples, or it might be for example that there just isn't enough information to solve the problem (8 parameters from 7 wavebands, remember).

If we suspected it was just that the solution was trapped in some local minimum because of an inappropriate starting point for the optimisation, we could include e.g. the lines:

```
[parameter.result]
filename = 'output/modis/MODIS_WW_1_A_1.params_a'
```

in the configuration file, as the optimisation would then start from the state found from the MERIS data, but in this case that is of little value.

So, we can 'fit' to the MERIS data for a single date very well, but the resultant state estimates have very high uncertainty, for some state elements simply because of poor spectral sampling. And we have some MODIS data with wider spectral sampling (though fewer wavebands) but we cannot make direct use of it, even with two observations on a single day.

The obvious thing to try then is to combine the observational datasets, and this is something we can do very easily in eoldas. Even though the spectral sampling of the data is very different, we can just define different observation operators (using the same radiative transfer model) to deal with this. Of course, there are other issues to consider, such as spatial sampling/location, but the same principle applies there in a general DA system.

We can base the new configuration on that we used for MERIS, and simply declare the MODIS operator with a different name: (confs/modis_single_b.conf)

```
# configuration file
# a minimal configuration file
# with overrides for meris_single.conf

[parameter.result]
filename = 'output/modis/MODIS_MODIS_WW_1_A_1.params_b'

# A New Operator
[operator]
obs_modis.name=Observation_Operator
obs_modis.datatypes = x,y

# NB -- give a new name to the meris operator output
[operator.obs.y.result]
filename = 'output/meris/MERIS_WW_1_A_1.fwd_b'

[operator.obs_modis.rt_model]
model=semidiscrete2
use_median=True
help_use_median = "Flag to state whether full bandpass function should be used or not.\nIf True, t
bounds = [400,2500,1]
help_bounds = "The spectral bounds (min,max,step) for the operator'
ignore_derivative=False
help_ignore_derivative = "Set to True to override loading any defined derivative functions in the

[operator.obs_modis.x]
names = $parameter.names[1:]
sd = [1.0]*len($operator.obs_modis.x.names)
datatype = x

[operator.obs_modis.y]
control = 'mask vza vaa sza saa'.split()
names = "465.6 553.6 645.5 856.5 1241.6 1629.1 2114.1".split()
sd = "0.003 0.004 0.004 0.015 0.013 0.01 0.006".split()
state = data/brdf_WW_1_A_1.kernelFiltered.dat
help_state='set the obs_modis state file'

[operator.obs_modis.y.result]
filename = 'output/modis/MODIS_WW_1_A_1.fwd_b'
```

We can then run with both meris and modis data:

```
eoldas_run.py --conf=config_files/eoldas_config.conf \
  --conf=config_files/meris_single.conf \
  --parameter.limits='[[232,232,1]]' \
  --conf=config_files/modis_single_b.conf
```

In any case, the result still does not have a good convergence, so the combination has not been a complete success.

The new (combined) state estimate is in `output/modis/MODIS_MODIS_WW_1_A_1.params_b`:

```
#PARAMETERS time gamma_time xlai xhc rpl xkab scen xkw xkm xleafn xs1 xs2 xs3 xs4 lad sd-gamma_tim
 232.000000    0.050000    0.295873    5.000000    0.010000    0.961493    0.484655    0.950000
```

The new `xlai` value is 0.435037 with a standard deviation of 0.000730. This corresponds to a LAI of 1.66 (confidence interval 1.58 to 1.66). The `xkab` value though is 0.958366 which is very low (4.25) when inverse transformed. The apparent uncertainty for this (sd) is 0.018995 on `xlai`, so the `xlai` confidence interval is 0.0 to 6.25, since it is bounded at 0. These posterior estimates are probably unreliable: we can suggest that because the optimisation routine reported that it did not find a very well converged solution.

A Coding note and caveat

Warning: This needs updating!

Note that in the configuration file we have specified the operator.obs_modis.rt_model.model field as rtmodel_ad_trans1, rather than rtmodel_ad_trans2.

In reality, these are the same models, but because they are FORTRAN shared object libraries containing the semidiscrete model and because they make use of global variables, one cannot run more than one observation operator that call the same shared object. To get around this problem, we have essentially made multiple copies of the library, so that different observation operators (e.g. using data from different sensors) can each have their own version. New operators in the future should be written with more careful consideration of such issues. Similar coding issues also make it next to impossible to run multi-core paralellisation optimisations to speed up code execution. The lesson is ... avoid globals like the plague, but there is normally some way we can get around the problem. In this case, the implication is that only 8 different observation operators calling the semidiscrete code can be run in one data assimilation exercise, as we have only generate 8 copies of the library.

Other forms of constraint

We have seen that we can apply eoldas to state vector estimation using the semidiscrete canopy reflectance model. This is very much what we want to be able to do from EO data: infer characteristics of the land surface, so that is positive. However, observations from a single sensor very often have insufficient information content to reliably allow an estimate of the range of state variables we know to affect the signal. The only constraint we have applied here beyond the observations (and the model) is that of physical bounds to some of the states (N.B. in eoldas, if you want to set only an upper or lower bound, you can use None to indicate that no bound should be set).

Even though we managed a 'stable' retrieval from MERIS data, the resultant uncertainties were very high, which makes the value of the retrieval rather low for any practical purposes. We saw that in this case, we could not find a viable solution for the 2 MODIS samples we used. Also, we demonstrated how to use the observation operator can be used to map from one EO measurement set (the 15 MERIS wavebands) to anoth (MODIS). The comparison was poor however because of the poorly constrained estimate of the states from the MERIS data.

How then can we improve the situation? The only real answers are: (i) get sensor data with higher information content (there are practical, physical and financial limitations to that however); (ii) add other sources of information. In generating global satellite products, for example, it is common practice to assume that some of the state variables (e.g. those affecting leaf and soil reflectance) are *known*. This is rather a dangerous assumption and it would be better to at least phrase this supposed insight into some distribution (applying hard physical bounds which we can implement in eoldas by changing the limits of validity of the states; or assuming some statistical distribution')

To demonstrate this, we build a new configuration file, `config_files/modis_single_c.conf`:

```
# configuration file
# a minimal configuration file
# with overrides for meris_single.conf

[parameter.result]
filename = 'output/modis/MODIS_MODIS_WW_1_A_1.params_c'
```

```
# A New Operator
[operator]
obs_modis.name=Observation_Operator
obs_modis.datatypes = x,y
prior.name=Operator
prior.datatypes = x,y

[operator.prior.result]
filename = output/modis/MODIS_MODIS_WW_1_A_1_prior_c.dat

[operator.prior.x]
names = $parameter.names[1:]
sd = [1.0]*len($operator.prior.x.names)
datatype = x

[operator.prior.y]
names = $parameter.names[1:]
sd = 0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5
state = 0.5,5,0.01,0.5,0.5,0.5,0.5,1.5,1.0,0.0,0,0,5
datatype = x

# NB -- give a new name to the meris operator output
[operator.obs.y.result]
filename = 'output/meris/MERIS_WW_1_A_1.fwd_c'

[operator.obs_modis.rt_model]
model=semidiscrete2
use_median=True
help_use_median = "Flag to state whether full bandpass function should be used or not.\nIf True, t
bounds = [400,2500,1]
help_bounds = "The spectral bounds (min,max,step) for the operator'
ignore_derivative=False
help_ignore_derivative = "Set to True to override loading any defined derivative functions in the

[operator.obs_modis.x]
names = $parameter.names[1:]
sd = [1.0]*len($operator.obs_modis.x.names)
datatype = x

[operator.obs_modis.y]
control = 'mask vza vaa sza saa'.split()
names = "465.6 553.6 645.5 856.5 1241.6 1629.1 2114.1".split()
sd = "0.003 0.004 0.004 0.015 0.013 0.01 0.006".split()
state = data/brdf_WW_1_A_1.kernelFiltered.dat
help_state='set the obs_modis state file'

[operator.obs_modis.y.result]
filename = 'output/modis/MODIS_WW_1_A_1.fwd_c'
```

The only new sections are:

```
[operator]
obs_modis.name=Observation_Operator
obs_modis.datatypes = x,y
prior.name=Operator
prior.datatypes = x,y

[operator.prior.result]
filename = output/modis/MODIS_MODIS_WW_1_A_1_prior_c.dat

[operator.prior.x]
names = $parameter.names[1:]
sd = [1.0]*len($operator.prior.x.names)
```

```
datatype = x

[operator.prior.y]
names = $parameter.names[1:]
sd = 0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5
state = 0.5,5,0.01,0.5,0.5,0.5,0.5,1.5,1.0,0.0,0,0,5
datatype = x
```

which state that the operator (called `operator.prior`) is to have $H(x) = I(x)$ (we call the operator Operator, the default) and we, simplistically here, set the state and standard deviation to 0.5. This then is a quite loose constraint as most states are bounded [0,1].

We run now with:

```
eoldas_run.py --conf=config_files/eoldas_config.conf \
  --conf=config_files/meris_single.conf \
  --parameter.limits='[[232,232,1]]' \
  --conf=config_files/modis_single_c.conf
```

Now, we get convergence, with the results in `output/modis/MODIS_WW_1_A_1.params_c`:

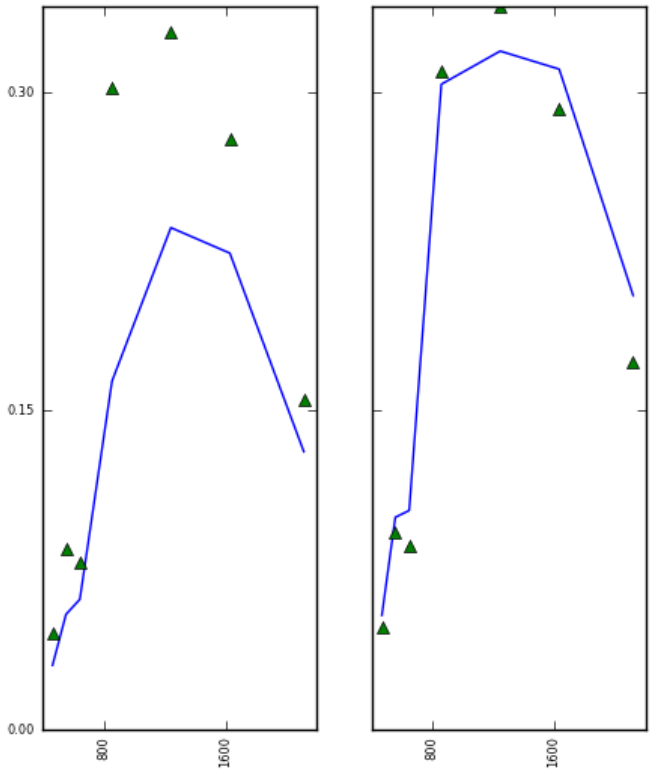
```
#PARAMETERS time gamma_time xlai xhc rpl xkab scen xkw xkm xleafn xs1 xs2 xs3 xs4 lad sd-gamma_tir
 232.000000    0.050000    0.295724    5.000000    0.010000    0.961483    0.484544    0.950000
```

One particular thing to note here is that *all* of the uncertainties are now less that 0.5. This happens (simply?) because we set the prior uncertainty to 0.5.

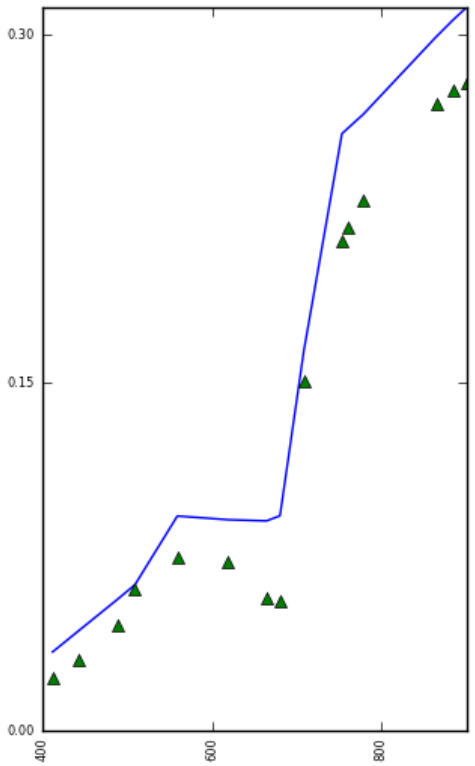
Note: Think about why that happens, statistically... essentially because $C_{post}^{-1} = C_a^{-1} + C_{prior}^{-1}$.

That is fine, and, as we have noted, expected (indeed we can think of that *being* the main purpose of applying the prior constraint), but it does mean that users of eoldas must bear in mind the consequences of setting too tight constraints as they will result in *apparently* low posterior uncertainties.

The original and forward modelled MODIS data resulting from this are:



and the MERIS:



The reproduction of the observations is clearly not wonderful, indicating some other issues with the data or model, but at least we have demonstrated how a prior can be used in eoldas to help condition the solution. As indicated above, one has to be careful applying prior constraints, as the results can be very dependent on the values chosen, but really such care must be taken about all choices here (which model, which data etc.).

Summary

In this section, we have moved from applying simple operators where the domain of the state vector and observations was the same to a more typical and useful case (for EO interpretation and use), where they are not. We have introduced a physically-based radiative transfer tool as the operator function (one that is provided with an adjoint for (relatively) rapid optimisation of the combined cost functions J.

We have shown how eoldas configuration files can be progressively modified (and used in combination) to set up particular problems. We have obttained an estimate of biophysical parameters from MODIS data (failed to converge for observations on a single date), from MERIS data, and from both MODIS and MERIS combined.

This is already quite a powerful tool, but we will see in the next section that other constraints can be interesting as well.

2.6 A synthetic experiment: simulating the performance of Sentinel-2

A useful application of the EOLDAS tools is to simulate how different sensors or combinations of sensors might be expected to work together. Simply, we could generate a set of artificial observations, acquired by a sensor (or sensors) from which the spectral, noise and orbital acquisition characteristics are known (or we can try different set ups to see their trade offs). We can then use EOLDAS to invert these synthetic observations and compare to the reality that went into them. This type of experiments can be very useful to test, for example, the usefulness of constellation concepts. The current example is a reduced version of the experiments that are presented in Lewis et al (2010), and show the use of EOLDAS in simulating ESA's upcoming Sentinel-2 performance for the inversion of biophysical parameters. In this document, we try to reproduce the middle column in Figure 2 in Lewis et al (2010)

The experiment can be split into a set of simple tasks:

1. **We need to generate an appropriate set of observations. On the one hand, this requires**
    - An idea of typical angular sampling.
    - An idea of the statistics of missing observations due to e.g. cloud.
    - Trajectories for the land surface parameters.
2. *Forward model* the land surface parameters to land surface reflectance.
3. Add noise to the observations
4. Invert the observations, with whatever prior information might be available.

The first step will require the definition of a typical re-visit schedule for the sensor, as well as information on illumination geometries. We make use of the pyephem Python package to calculate the solar position at a given latitude and longitude. Some simple rules are given to model the sensor acquisition geometry (for example, the view zenith angle is assumed to be random). We can use full wavebands, or just the median wavelength of each band (the latter is far more efficient, and this solution can be used as a starting point to solve the full bandpass problem).

A further refinement is the inclusion of missing observations due to cloud cover, for example. It is typically observed that cloudiness is correlated in time, so a model of cloudiness that simulates the typical burst nature of cloud statistics is required. Another way to go about this is to use long term cloudiness observations, but we choose a very simple approach in this work.

The parameter trajectories are functions that describe the evolution of a given parameter in time. In this case, we choose to vary the following parameters:

- LAI
- Chlorophyll concentration
- Leaf water
- Leaf dry matter
- Number of leaf layers
- First soil component

This is an ambitious scenario: the analytic trajectories chosen for these parameters are smooth, but with very different degrees of smoothness. Some parameters are set to be constant (leaf layers and leaf dry matter). We will assume that the regulariser that solves this problem is identical for all the parameters, and constant in time. Even though we know that this is not the case, this simplifying assumption still allows the recovery of reasonable parameter estimates. Note that while in some cases (such as leaf area index) one might have mechanistic models that maybe use meteorological inputs to simulate photosynthesis and allocation of assimilated carbon to the leaves to estimate the amount of leaf area, for other parameters that have a significant impact on the remote sensing signal, these models might not be available.

The forward modelling of the parameter trajectories, coupled with the required wavebands and illumination geometries, produces a set of synthetic observations: this is what a sensor, operating with the given characteristics would "see", in terms of surface reflectance. As with any measurement, noise needs to be incorporated. We do this by adding some Gaussian noise to the simulated reflectances, in line with what we expect to be typical values after atmospheric correction. Although we assume noise to be uncorrelated across bands, in practice it will be.

Finally, we solve the inverse problem: find the parameter trajectories given the set of noisy and incomplete observations. In reality, we also need to estimate the strength of the regularisation, a hyperparameter. In Lewis et al (2011) this is done using cross-validation. In the paper, we proceeded as follows:

1. Set a number of values that Γ might take. This will necessarily be a large range (say from $10^{-5} - 10^{5}$), but prior experience might dictate a suitable interval.
    (a) For one value of Γ...
        i. Select one single observation, and remove it from the observations that will go into the inversion
            A. Solve the inverse problem
            B. Predict the missing observation, and compare with the truth
        ii. Calculate a prediction metric, such as root mean square error (RMSE) or similar
2. Select the value of Γ that results in the most accurate predictions.

This method is clearly very time consuming, but can be carried out in parallel. Note that the optimal value of Γ will be different if the nature of the problem (priors, order of the smoothness constraint, etc) changes. We shall not concern ourselves with crossvalidaiton here, and will use the optimal values from the paper.

For comparison purposes, we will also solve each observation independently, which is similar to what one would do in look-up table-based inversion approaches. In general, the problem of inverting six parameters from a single multispectral observation is incredibly ill-posed. The very informative sampling scheme of the MSI2 sensor however allows inversions. We shall also calculate the associated uncertainty of these single inversions, which we expect to be very high (and exhibit large degrees of correlation between parameters), and also note that when no observations are available, there will be no estimate of land surface parameters.

The next sections describe the python script that performs the experiment.

2.6.1 Anatomy of the simulation code

The code is organised in a single class, `Sentinel`. This class will perform all the above described tasks. The user might want to modify this class to perform other experiments. The class basically writes out files, and also runs the eoldas prototype as required. As mentioned above, we use the pyephem package to simulate the solar geometry.

2.6.2 Generating the synthetic observations

The code for this is fairly simple, and can be seen in the `main` function in `sentinel.py`:

```
lad = 1,5


[general]
is_spectral = True
calc_posterior_unc=True
help_calc_posterior_unc="Posterior uncertainty calculations"
write_results=True
doplot=True
```

We first instantiate the `Sentinel` class, calculate the parameter temporal trajectories using the `Sentinel.parameters` method. This requires a time axis as well as an output file (in this case, `input/truth.dat`. This file can also be used to compare inversion results etc.). Once the parameter trajectories have been established, we can forward model them to surface reflectance using the `Sentinel.fwdModel` method. This method has a number of options that are important to note:

`ifile` input data (parameters) file

`ofile` output reflectance data file

`lat` latitude (default '50:0') (see ephem)

`lon` longitude (default "0:0') (see ephem)

`year` as int or string (default '2011')

`maxVza` maximum view zenith angle assumed (15 degrees default)

`minSD` minimum noise

`maxSD` maximum noise. The uncertainty in each waveband is scaled linearly with wavelength between minSD and maxSD

`fullBand` set True if you want full band pass else False (default False). Note that its much slower to set this True.

`confFile` configuration file (default `config_files/sentinel0.conf`). If this file doesnt exist, it will be generated from `self.confTxt`. If that doesn't exist, `self.generateConfTxt()` is invoked to provide a default.

`every` sample output every 'every' doys (default 5)

`prop` proportion of clear sample days (default 1.0)

`WINDOW` size of smoothing kernel to induce temporal correlation in data gaps if prop < 1

Note that some other settings (such as default parameter values, and which parameters will be varied temporally) are set in the class constructor. In this case, we run the forward model twice, to create two distinct datasets: a complete best-case scenario, and a second scenario where the missing observations (with a proportion of 0.5 missing observations). These two datafiles are "clean", i.e., there's no noise. The `Sentinel.addNoiseToObservations` adds Gaussian independent noise to the observations. The variance of the noise is estipulated in the header of the clean data file (a linear increase of uncertainty with wavelength).

2.6.3 Solving each data individually

This part of the code will solve for each observation individually: assuming some prior knowledge, we solve for the weighted least squares fit. Mathematically, this for each observation $\mathbf{y}$, we minimise the functional $J(\mathbf{x})$, where $\mathbf{x}$ is the state tht describes that observation.

$$J(\mathbf{x}) = (\mathbf{x} - \mathbf{x}_p)^T \mathbf{C}_{prior}^{-1} (\mathbf{x} - \mathbf{x}_p) + (H(\mathbf{x}) - \mathbf{y})^T \mathbf{C}_{obs}^{-1} (H(\mathbf{x}) - \mathbf{y})$$

Solving for single observations is a very hard problem: there usually isn't an unique solution as parameter compensate for each other, and the observational constraint does not have enough information to allow this. This is made even worse by additional noise. In consequence, the shape of $J(\mathbf{x})$ is very flat over large areas, showing no strong preference for values of the state vector. As a first test, we can use the noise-free synthetic observations and invert them using some kind of gradient descent algorithm that minimises $J(\mathbf{x})$. To make it faster, we can start the gradient descent algorithm with the true values, and then calculate the Hessian and its inverse and look at uncertainties in retrieved parameters. This is quite instructive in its own right. A second test would be start the minimisation at some other point that it's not the true state, and see how the solution compares. Finally, we'd want to invert each individual observation, taking into account the noise.

We test some of these ideas with the clean datasets (those that have had no noise added to them). We also plot the results using graphical methods `Sentinel.crossPlot` and `Sentinel.paramPlot`.

```
plotmod=30
help_plotmod='frequency of plotting'
plotmovie=False
epsilon=10e-15
help_epsilon="Epsilon"

[general.optimisation]
randomise=False

[operator]
prior.name=Operator
prior.datatypes = x,y
obs.name=Observation_Operator
obs.datatypes = x,y

[operator.prior.x]
names = $parameter.names[1:]
datatype = x

[operator.prior.y]
control = 'mask'.split()
names = $parameter.names[1:]
sd = [10.0]*len($operator.prior.y.names)
help_sd='set the prior sd'
datatype = y
state = $parameter.x.default[1:]
help_state = "Set the prior state vector"

[operator.prior.y.result]
filename='output/rse1/rse1_test_prior.dat'
help_filename = 'prior filename'

[operator.obs.rt_model]
model=semidiscrete1
use_median=True
help_use_median = "Flag to state whether full bandpass function should be used or not. If True, t
bounds = [400,2500,1]
help_bounds = "The spectral bounds (min,max,step) for the operator'
ignore_derivative=False
help_ignore_derivative = "Set to True to override loading any defined derivative functions in the

[operator.obs.x]
```

2.6.4 Solving using data assimilation

Finally, we can solve the problem using the DA framework. This is done using the `Sentinel.solveRegular` method. One way to speed up processing is to start the inversion with the results calculated above for the single observations.

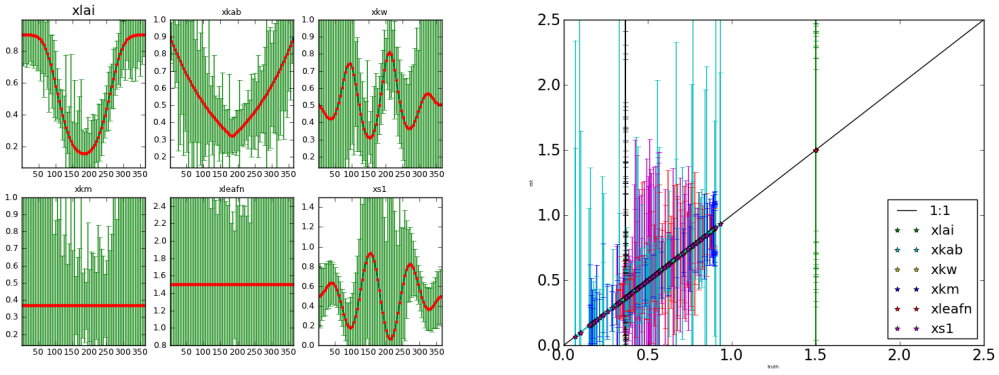


Figure 2.1: Inverting each individual observation (no noise) starting from the true solution. Left panel: (transformed) parameters and 95% CI Right panel: scatterplot of retrieved parameters vs true parameters.

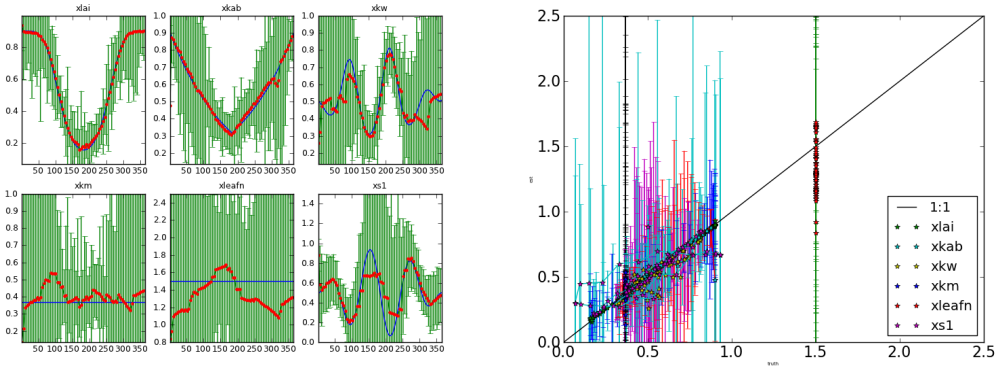


Figure 2.2: Inverting each individual observation (no noise) not specifying the true solution. Left panel: (transformed) parameters and 95% CI Right panel: scatterplot of retrieved parameters vs true parameters.

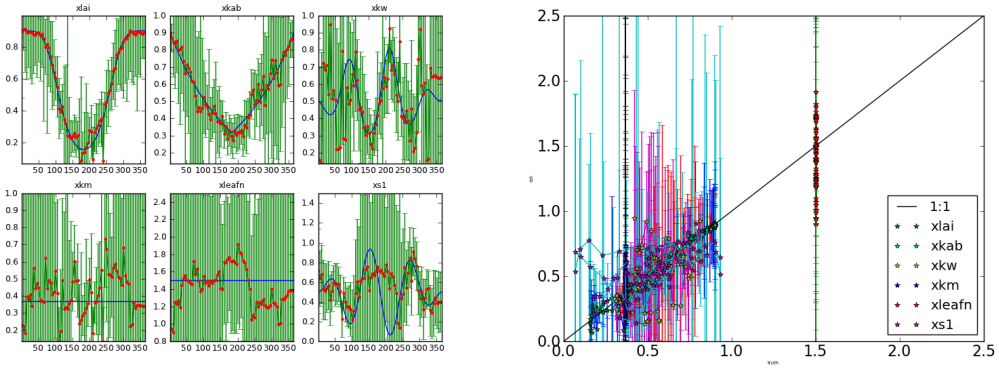


Figure 2.3: Inverting each individual noisy observation starting from the true solution. Complete series. Left panel: (transformed) parameters and 95% CI Right panel: scatterplot of retrieved parameters vs true parameters.

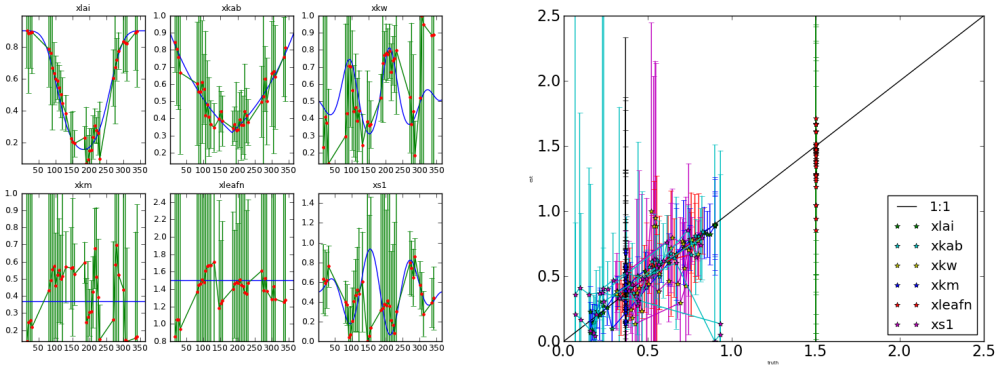


Figure 2.4: Inverting each individual noisy observation starting from the true solution. Gappy series. Left panel: (transformed) parameters and 95% CI Right panel: scatterplot of retrieved parameters vs true parameters.

```
sd = [1.0]*len($operator.obs.x.names)
datatype = x

[operator.obs.y]
control = 'mask vza vaa sza saa'.split()
names = ['433-453','457.5-522.5','542.5-577.5','650-680','697.5-712.5','732.5-747.5','773-793','78
sd = ["0.004", "0.00416142",  "0.00440183", "0.00476245", "0.00489983", "0.00502003","0.00516772",
datatype = y
state = 'data/rse1_test.100.dat'
help_state='set the obs state file'

[operator.obs.y.result]
```

Note: Here we need to put the solution plots. However, it appears that we might need new Γ values?

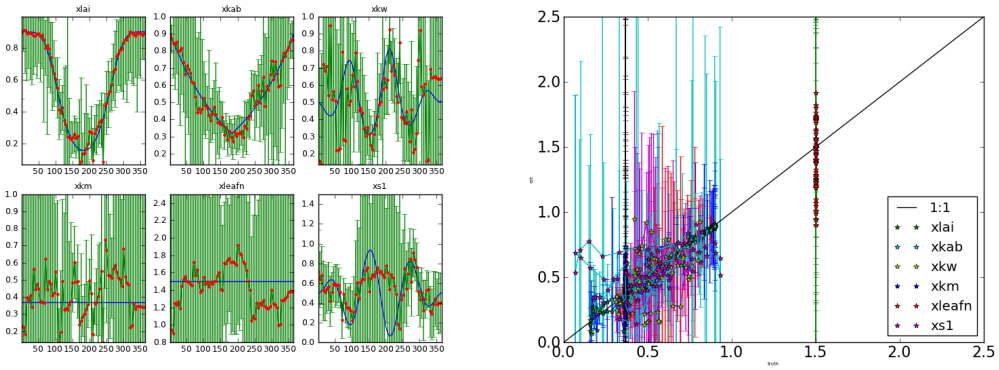


Figure 2.5: Inverting each individual noisy observation starting from the true solution. Complete series. Left panel: (transformed) parameters and 95% CI Right panel: scatterplot of retrieved parameters vs true parameters.

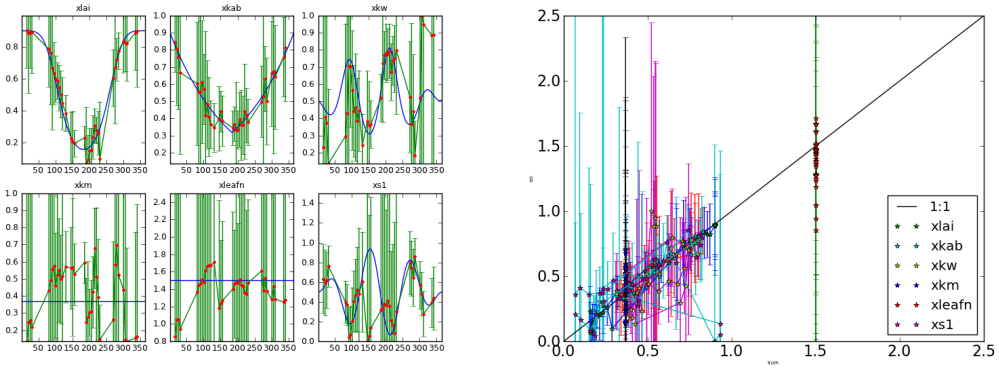


Figure 2.6: Inverting each individual noisy observation starting from the true solution. Gappy series. Left panel: (transformed) parameters and 95% CI Right panel: scatterplot of retrieved parameters vs true parameters.

2.7 Inverting a time series of MODIS observations over agricultural fields

2.7.1 Introduction

The current example builds on the previous synthetic example by using data from the MODIS sensor on the TERRA and AQUA platforms to invert the state of the land surface over an agricultural area in Thuringia (Germany). We shall use a script similar to sentinel.py but will use a slightly different inversion strategy. The data are in the file brdf_WW_1_A_1.kernelFiltered.dat

The solution strategy tries to overcome one of the weaknesses of the system so far: the need for very costly cross-validation in order to estimate the hyperparameters. This is done by using the single observation estimates (assuming that they are a reasonable approximation to the true estate) to initialise the EOLDAS inversion. Since the single observation estimates are usually gappy, we interpolated them smoothly using the regularisation method introduced in Garcia (2010) This step is effectively a fast equivalent of the smoothing of NDVI time series, that includes generalised cross validation. This results in smooth time series of parameters, as well as estimates of the regularisation hyperparameter (for each of the components of the state vector). Starting the final EOLDAS inversion from here, using a value of the hyperparameter derived from the smoothing puts the cost function minimiser close to the minimum, so fewer iterations are needed to solve the problem.

2.7.2 The script

The script is quite simple, and uses the `Sentinel` class from the Sentinel synthetic experiment. The code is stored in a function called `main`:

```
def main (ifile = 'data/brdf_WW_1_A_1.kernelFiltered.dat', \
        confFile='config_files/sentinel_Def.conf', \
        solve = ['xlai','xkab','scen','xkw','xkm','xleafn','xs1']) :
  '''
  Show that we can use this same setup to solve
  for the field data (MODIS) by using a different config file
  '''


  ifile = 'data/brdf_WW_1_A_1.kernelFiltered.dat'
  ofileSingle = 'output/brdf_WW_1_A_1.kernelFilteredSingle.dat'
  ofile = 'output/brdf_WW_1_A_1.kernelFiltered.dat'

  s = Sentinel(solve=solve,confFile=confFile)
  # as above, solve for initial estimate
```

```
s.solveSingle(ifile,ofileSingle)
s.paramPlot(s.loadData(ofileSingle),s.loadData(ofileSingle),\
                filename='plots/testFieldDataSingle.png')
s.smooth(ofileSingle,ofile=ofileSingle+'_smooth')
s.paramPlot(s.loadData('input/truth.dat'),\
            s.loadData(ofileSingle+'_smooth'),\
            filename='plots/%s.png'%(ofileSingle+'_smooth'))

gamma = np.median(np.sqrt(s.gammaSolve.values()) * \
      np.array(s.wScale.values()))
gamma = int((np.sqrt(s.gammaSolve[solve[0]]) * \
      np.array(s.wScale[solve[0]]))+0.5)
s.solveRegular(ifile,ofile,modelOrder=2,gamma=gamma, \
      initial=ofileSingle+'_smooth')
s.paramPlot(s.loadData(ofileSingle+'_smooth'),\
            s.loadData(ofile + '_result.dat'),\
            filename='plots/%s_Gamma%08d.png'%(ofile,gamma))
```

The structure of the code will be familiar from the sentinel experiment: the only novelty is the use of the `Sentinel.smooth` method to perform the smoothing of the parameters. Instead of starting from the prior or from the true data, as we did in the synthetic experiment, we use the `initial` keyword to feed the smoothed interpolated estimates. We can see that the result of the single observation inversions are very noisy and with large uncertainties (file `plots/testFieldDataSingle.png`):

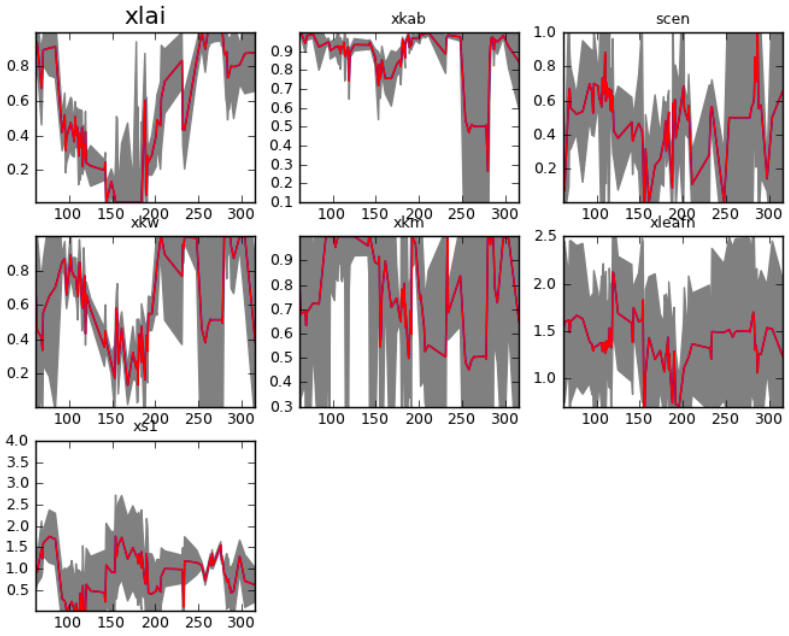


Figure 2.7: Results inverting each observation individually for a wheat field. Note that this plot is in transformed coordinates

The parameter-by-parameter smoothing using the method of Garcia (2010) results in a smoothed and complete version of the parameter evolution. We can see that this is already a fairly good estimate of the parameters. (the plot is in `plots/output/brdf_WW_1_A_1.kernelFilteredSingle.dat_smooth.png`)

Using this a starting value, we then solve the EOLDAS problem. We expect that this solution is already close the minimum, so solving the problem will be reasonably fast. We also use the Γ from the optimal smoothing/interpolation. While this is not exactly the same problem that we are solving in EOLDAS (different boundary conditions and limited to a second order differntial operator), we assume that there's tolerance to Γ (see Lewis et al 2012). The results confirm that we only tweak the starting solution slightly: the red

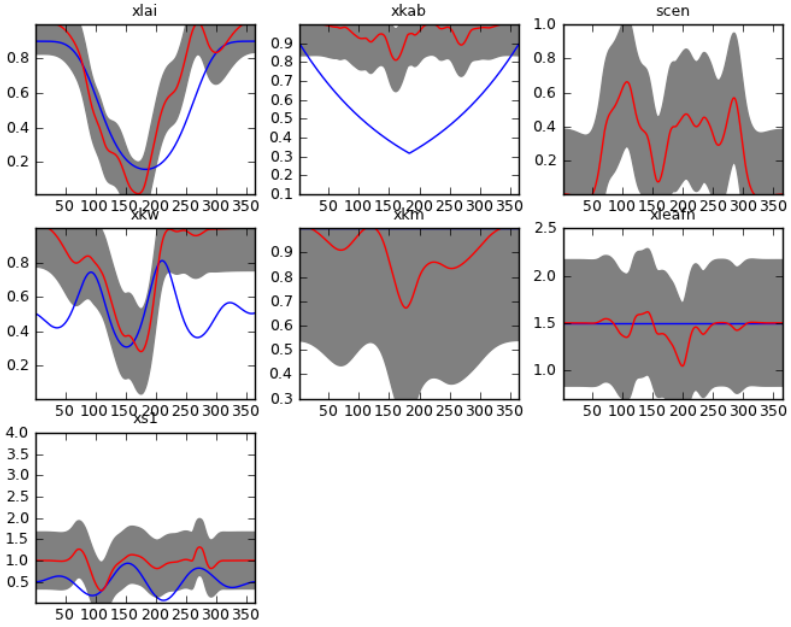


Figure 2.8: Parameter-by-parameter smoothing and interpolation. The red line shows the maximum a posteriori paremeter estimate, and the grey lines show a measure of the uncertainty. The blue line show the temporal evolution of the Sentinel synthetic experiment trajectories for comparison. Note that this plot is in transformed coordinates

line (with associated grey 95% CI) is very similar in most cases to the initial value (blue line). The plot is in `plots/output/brdf_WW_1_A_1.kernelFiltered.dat_Gamma00000529.png`:

The temporal evolution shown in this las plot is consistent with the development of a wheat field, and even senescence and leaf water dynamics can arguably be identified after the peak LAI period.

2.7.3 Some comments

This is nearly equivalent for solving for the dynamic model term. Since the solution is already quite close to the minimum, only slight modifications in the observation component are envisaged. Further speed-ups can be envisaged by using a linear approximation to the observation operator at this stage. Also, in a practical sense, the single inversions don't uncertainty calculations, and a faster first pass using look-up tables might add further speed to the whole process.

2.8 Spatial and multi-scale data assimilation examples

2.8.1 Introduction

Although only temporal data assimilation (DA) has been explored in depth in the prototype EO-LDAS tool, it was noted in Lewis et al. (2012a,b) that the tool should be capable of spatial, as well as temporal DA. The purpose of this technical note is to demonstrate the spatial capabilities of the tool and to show how multi-(spatial) resolution DA can be achieved.

To illustrate these concepts, we develop from the regularisation of NDVI data example in the EO-LDAS tutorial (www1). In this example, we demonstrate the concept of using regularisation (by a zero-order process model) to smooth and interpolate a noisy data sequence (that we suppose to be NDVI). The data in the experiment are synthetic, i.e. generated from a known truth. The experiment is run from a python code (www2) that generates

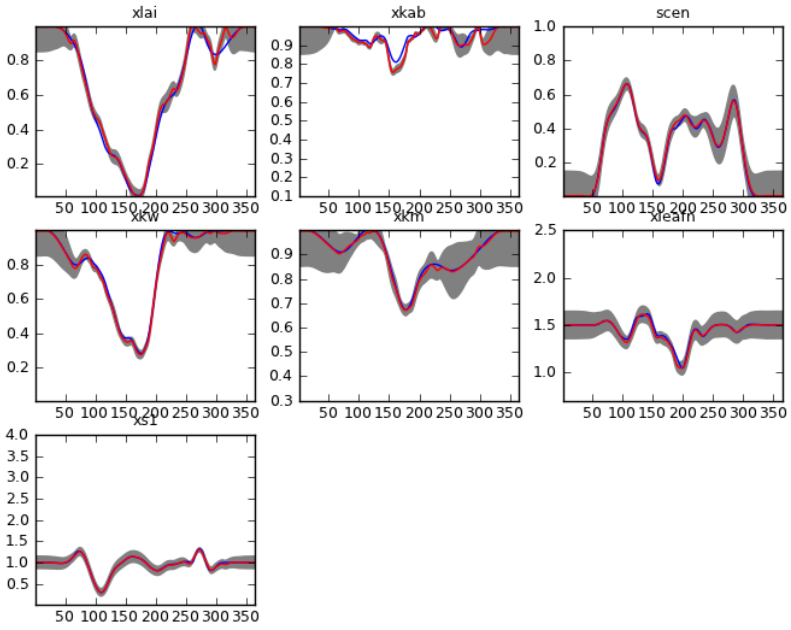


Figure 2.9: DA solution in red (plus grey 95% credible interval). The blue line shows the initial guess. Note that this plot is in transformed coordinates.

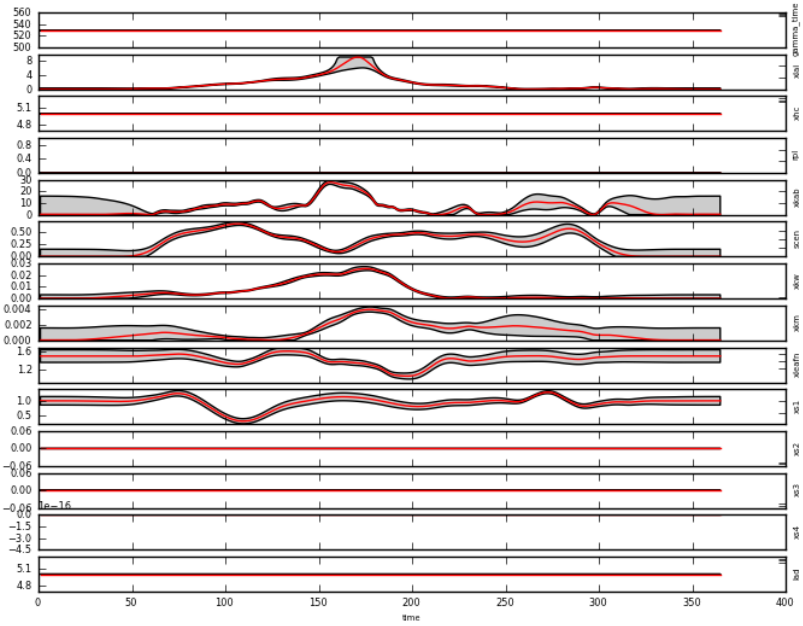


Figure 2.10: DA solution in red (plus grey 95% credible interval) in real coordinates . This plot can be found in `output/brdf_WW_1_A_1.kernelFiltered.dat_result.dat.plot.x.png`

the synthetic dataset, adds a significant amount of noise (standard deviation 0.15). Correlated gaps (mimicking clouds) are introduced into the data. In the example given 33% of the observations are removed. The results are shown in the figure below.

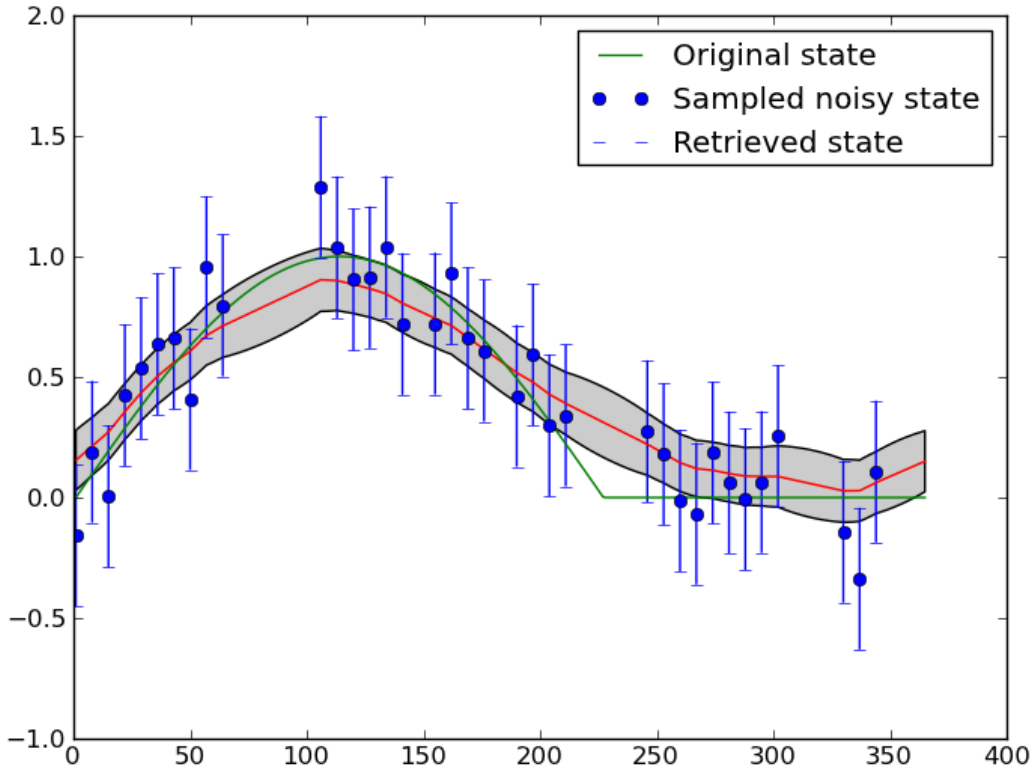


Figure 2.11: Fig 1: Smoothing a noisy time series with temporal DA.

The blue dots show the input data, with associated uncertainty (1.95 σ, i.e. the 95% confidence interval (C.I)). The 'truth' a half sine wave with a flat section is shown as the green line. The retrieved state is shown as the red line (mean) and grey bounds (95% C.I.). This is a suitable example to demonstrate the principles of temporal data assimilation with two constraints: (i) the noisy observations; and (ii) a simple (zero-order process model). Although some problems are encountered at sudden changes (the transition between the sine wave and the flat time), the form of the underlying function is well reconstructed from the noisy samples. In this case, the observation operator, i.e. the operator that translates between the space of the state we are trying to estimate here (NDVI) and the observations (NDVI) is an identity operator. This is chosen to make a fast experiment and to demonstrate principles. There is no attempt to optimise the 'smoothness' (gamma) term in this and following experiments. Instead, we take a theoretical value from the truth and reduce it by a factor of 5 (so we should generally be under-smoothing).

The EO-LDAS tool is accessed in this example through (i) a configuration file; (ii) a command-line to override some of the settings in a generic configuration file. The command line is set as:

[CODE]

and the (generic) configuration file given in (www3. In the configuration file, we declare the 'location' specification to be 'time' and specify the parameters we wish to solve for (NDVI here).

[CODE]

and set up the differential operator (in time):

[CODE]

and the (Identity) observation operator: [CODE]

In further examples in the tutorial and in Lewis et al. (2012a,b) we go on to show how more complex observation operators (such as a radiative transfer model) can be used in the DA so that vegetation state variables (LAI etc.) can be estimated from remote reflectance observations. The principle of the underlying (zero- or first-order) process models is the same as in the simple NDVI example.

The example is also interesting in its own right, as a demonstration of optimal filtering, i.e. smoothing with a target, multi-constraint cost function. It also shows how uncertainty can be calculated in such an optimal estimation framework.

2.8.2 Spatial data assimilation

Spatial data assimilation proceeds in exactly the same way as temporal DA when we use these simple process models. In fact, instead of considering the x-axis in figure 1 as time, we could simply state that it is a transect in space and all of the same results and conclusions would apply. In the spatial sense, we might consider this to be quite similar to what is done in optimal spatial interpolation schemes such as kriging or regression kriging (where some low order model is fitted and the difference constraint operates on the residuals of that model) (www4).

Python code for performing a spatial DA is given in www5. It is very much based on that for the temporal DA discussed above, but now we generate a synthetic dataset in two dimensions:

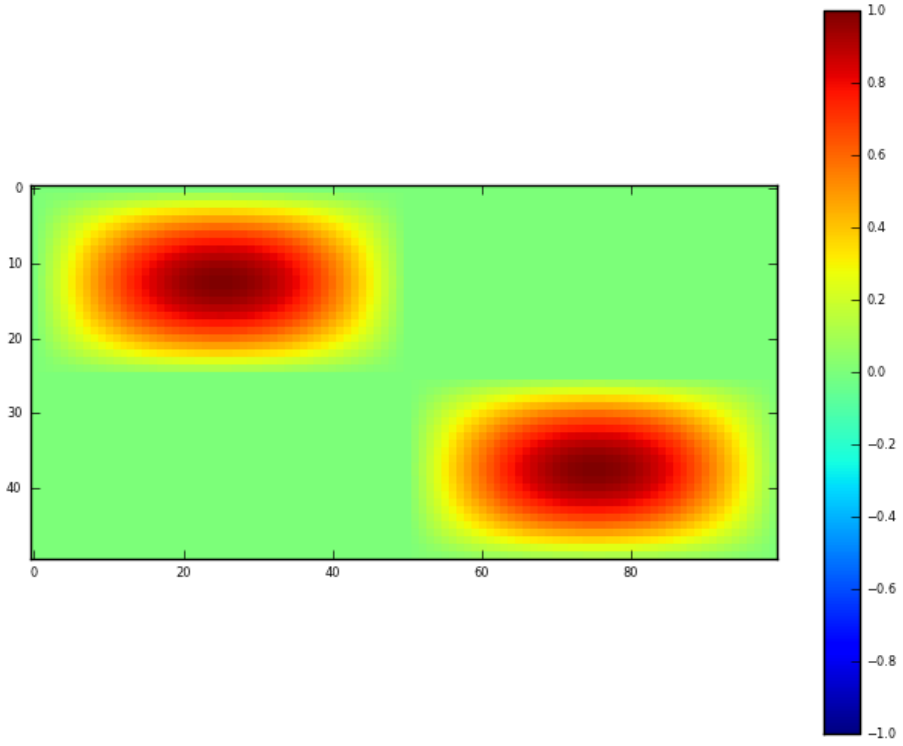


Figure 2.12: Fig. 2: Synthetic 2D (spatial) dataset

Figure 2 shows the spatial dataset generated in this code. It contains some 2-D sine waves and flat areas. The test data lies between the values 0 and 1 and represents NDVI, which one could imagine as representing say 4 'fields' here, two with crops in and two bare soil. In such an imagining, we can see spatial variation in the NDVI (vegetation density) within the fields. Noise, of sigma 0.15 is added to these data, and again 33% of the samples removed. The input dataset then is shown in figure 3.

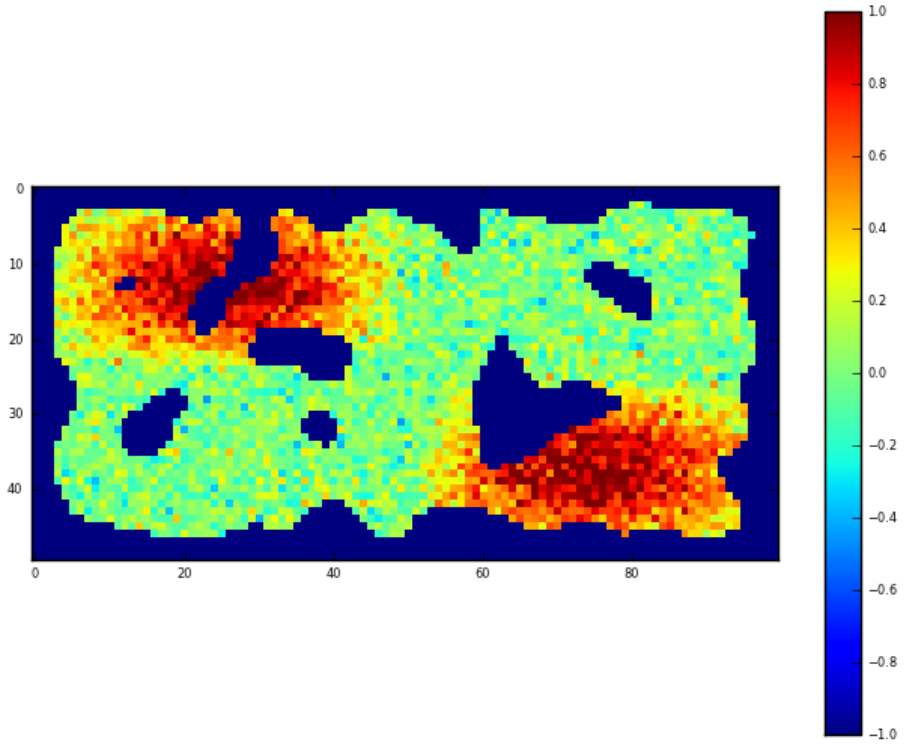


Figure 2.13: Fig. 3: Sampled synthetic 2D (spatial) dataset

We can again suppose the 'holes' in the observations to be representative of clouds. The configuration file for this experiment is given in *www6 <https://github.com/jgomezdans/eoldas_examples/blob/master/config_files/IdentitySpatial.conf>*. In setting up a spatial DA in EO-LDAS, we declare the location to be 'row' and 'col':

[CODE]

We then define two differential operators, one in row and the other in column space:

[CODE]

The observation operator is as in the previous example. The result of running the DA is an estimate of the NDVI for all sample locations:

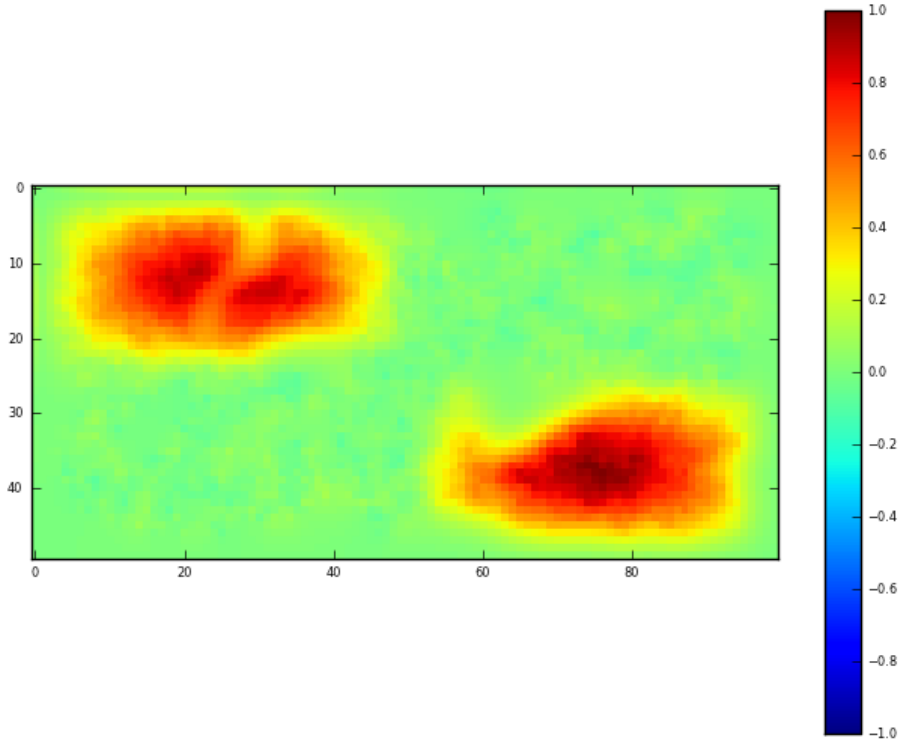


Figure 2.14: Fig. 4: Result of DA (mean)

which is our posterior estimate of NDVI obtained from the samples given in figure 3. It is our 'optimal estimate' of the original data (figure 2) and does a very reasonable job of this. The uncertainty in this estimate is given by:

Recall that the uncertainty in the input data was 0.15. Where we have sampled data, this has been reduced (by the DA/regularisation) to around 0.06, and in the gaps (i.e. under the clouds) the uncertainty is around 0.075. The fidelity of this reconstruction is perhaps better illustrated by taking a transect through the dataset:

The input data are shown as green dots, and the reconstruction given as the red line, with cyan (95%) C.I.s. The original data (i.e. what we are trying to reconstruct) is the blue line. As with the temporal example in figure 1, this does a remarkably good job. A scatterplot of the true (x axis) and retrieved (y-axis with 95% CI as green errorbars) is shown in figure 7:

The scatterplot reveals a slight bias in the retrieved NDVI for high NDVI values, which is probably a result of the small number of high values in the input dataset and the type of smoothing used). It may just be a feature of the assumption of stationarity in the smoothness term. If you compare the high NDVI values in figures 2 and 4

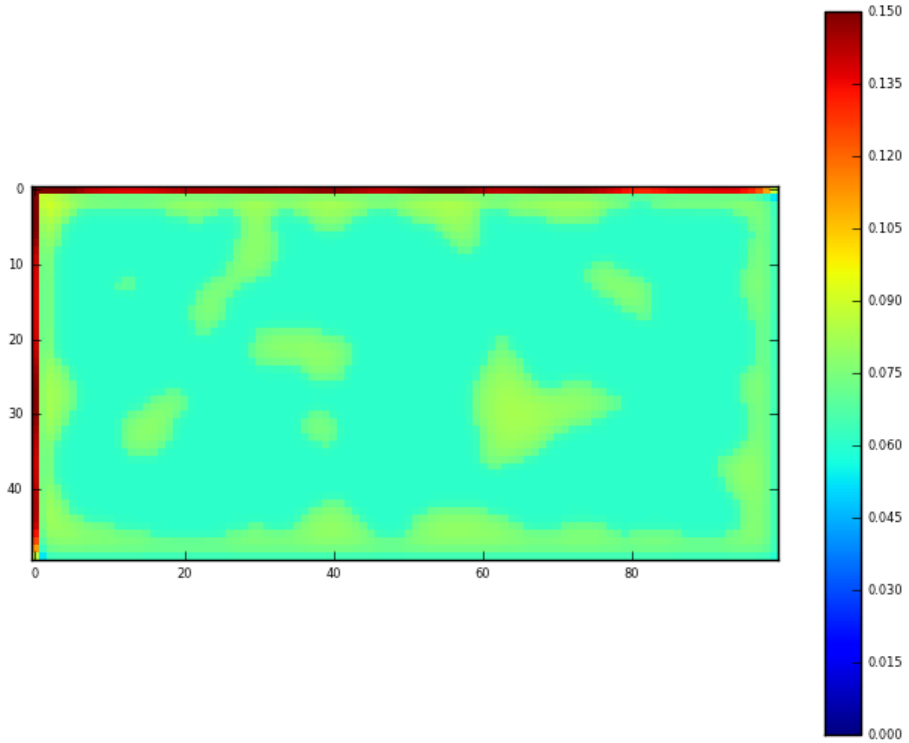


Figure 2.15: Fig. 5: Result of DA (sd)

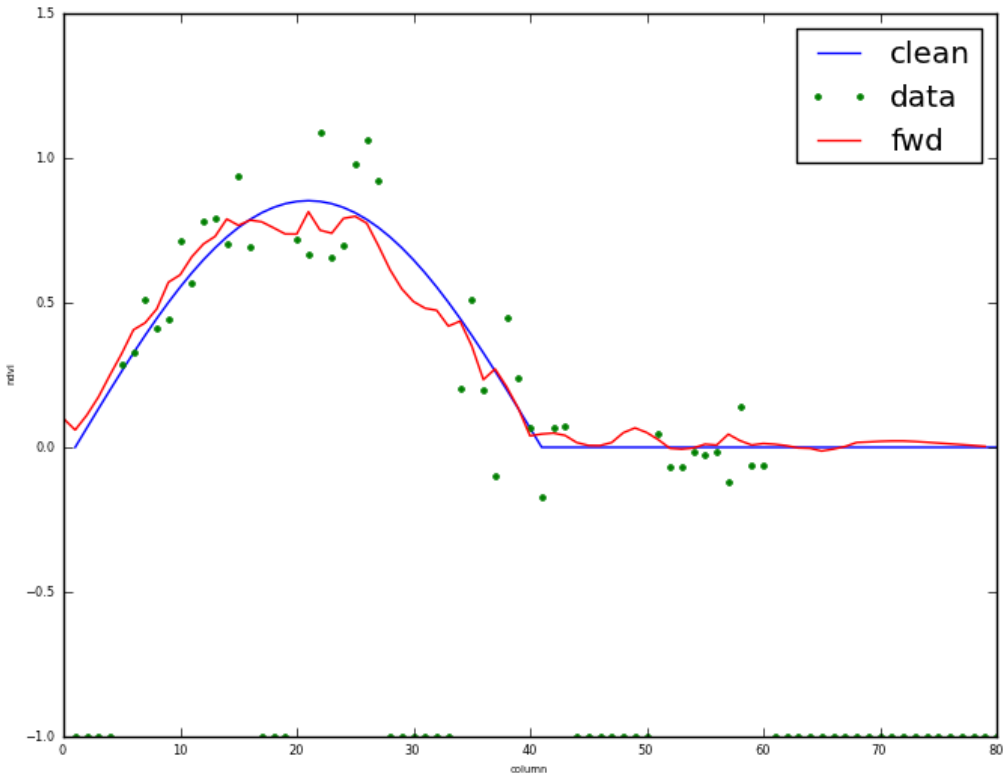


Figure 2.16: Fig. 6: Transect through dataset at row 13

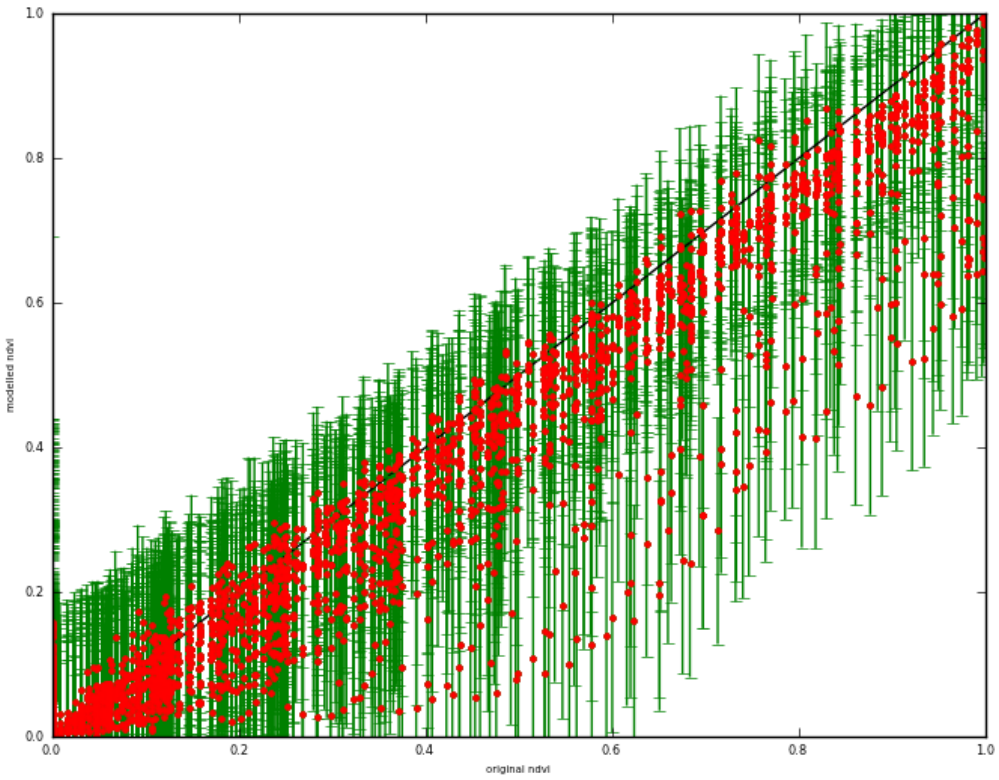


Figure 2.17: Fig. 7: Scatterplot of retrieved (y-axis) against true (x-axis) NDVI over all spatial samples

you can see this same issue, although it is relatively minor in the grand scheme of things. Certainly the 95% C.I. covers the extent of the true data, so the C.I. is likely slightly over-estimated here.

2.8.3 Multi-resolution data assimilation

We can proceed from this example to consider multi-spatial resolution DA within EO-LDAS. Although we do not have any sensor spatial transfer functions within the prototype, we can demonstrate and explore the principles within the existing tool. This can be done by simply mapping a coarser spatial resolution dataset to the grid of a higher resolution dataset. To account for the fact that the sample observations will then be used multiple times within the existing DA, we can simply inflate the apparent uncertainty of each sample that we load.

Code to achieve this is given in www7. In this, we generate two datasets, one at 'full' resolution, with an uncertainty of 0.15 and with 33% of the observations missing, and one at a linear scale of 1/4 th, i.e. where 16 pixels at high resolution represent one pixel at coarse resolution. The filter window size used to correlate the data gaps is 3 in this example (larger filter sizes will result in larger gaps). The uncertainty in the coarse resolution data is 0.0375, so less than that at high resolution (by a factor of 4) but then we re-inflate it to an apparent uncertainty of 0.15 when applying the same (coarse) resolution sample pixel over the high-resolution grid.

As in other examples in EO-LDAS, we use separate observation operators for the different data streams, though this is largely for convenience in this case as the both data sets are associated with Identity operators in this case. These results demonstrate the ability of the code to achieve a multi-resolution DA (albeit with a simple Identity observation operator here). With 1/3rd of the samples missing, the results are very good, although we note that the specifics of the 'gap' algorithm used here mean that gaps tend to be created at the edge of the image first (this is to do with how a random noise field is filtered to create the gappiness). There is no apparent bias in the results (figure 8f), and effective use is made of both the high- and low-resolution datasets to provide a viable (and in this case accurate) posterior estimate (figure 8b).

In a second example, www8, we consider the case where 2/3 of the data are missing, with a larger filter size (6) resulting in larger gaps. The results are clearly of somewhat lower quality, but this is reflected in the uncertainties. The uncertainty map (figure 10e) clearly demonstrates where the sampling in the input data (in both high and low resolution datasets) is poor (light blue). Unsurprisingly, where we have samples in both high and low spatial resolution datasets, the uncertainty is lowest. Given the amount of extrapolation in this exercise, the results are remarkably good. There is no apparent bias in the results (visually, from figure 10f). The transect in figure 11 shows that though the reconstruction is still perhaps a little noisy (it could most likely tolerate a higher gamma) it provides a faithful reconstruction of the original data from noisy multi-resolution datasets with large gaps.

In a final example (www9), we remove 2/3rd of the samples from the high-resolution image, but only 1/3rd of the lower resolution data. This is an attempt to mimic the impact of higher frequency low spatial observations with occasional high-resolution data (though we do not directly consider the time dimension in this example). In this case, we have only sparse coverage at high resolution, but good coverage of most of the major features at low resolution. There is minimal 'blockiness' in the DA result in figure 12b, but even this does not seem very apparent in the transect (figure 13). The better coverage provided by the low resolution data produces much less scatter when comparing to the original signal (compare figures 10f and 12f). The result compares very favourably with that in figure 8 which had twice as many high resolution samples.

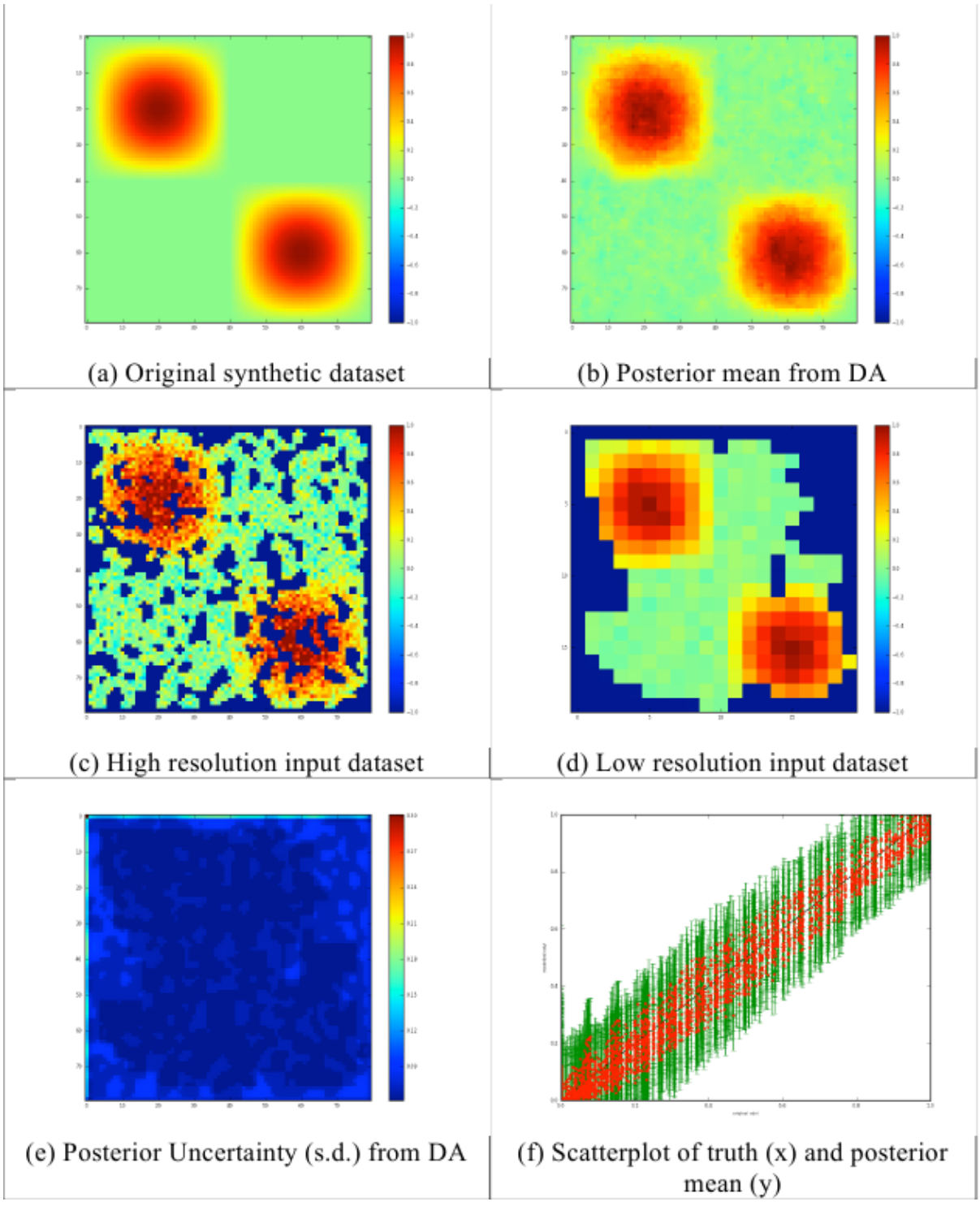


Figure 2.18: Fig. 8: Results of multi-scale analysis for 1/3 data missing

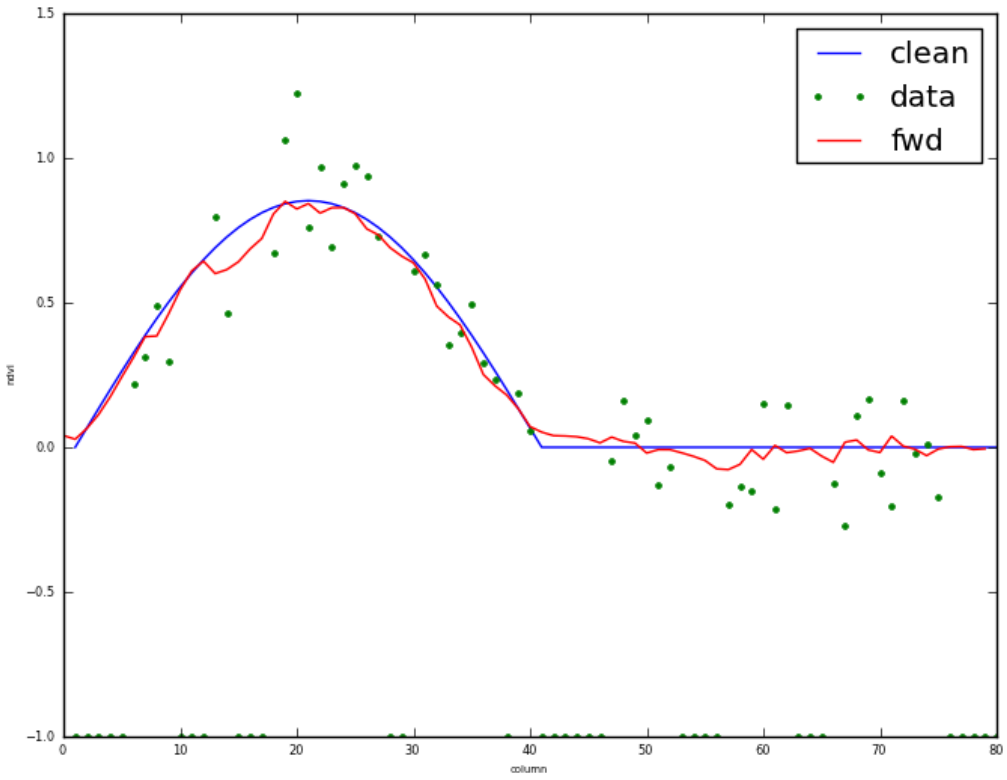


Figure 2.19: Fig. 9: Transect through row 13 of results for 1/3 data missing

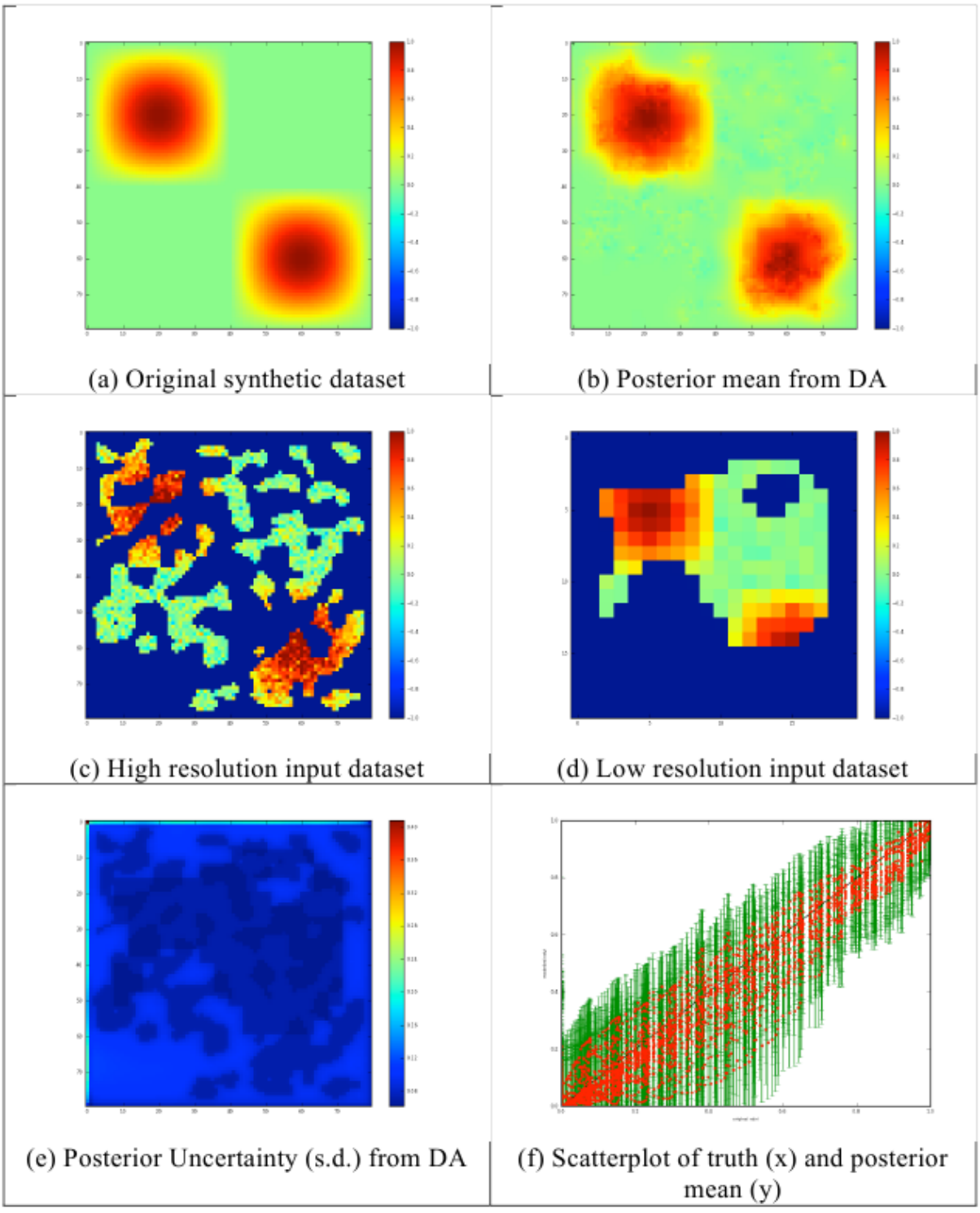


Figure 2.20: Fig. 10: Results of multi-scale analysis for 2/3 data missing

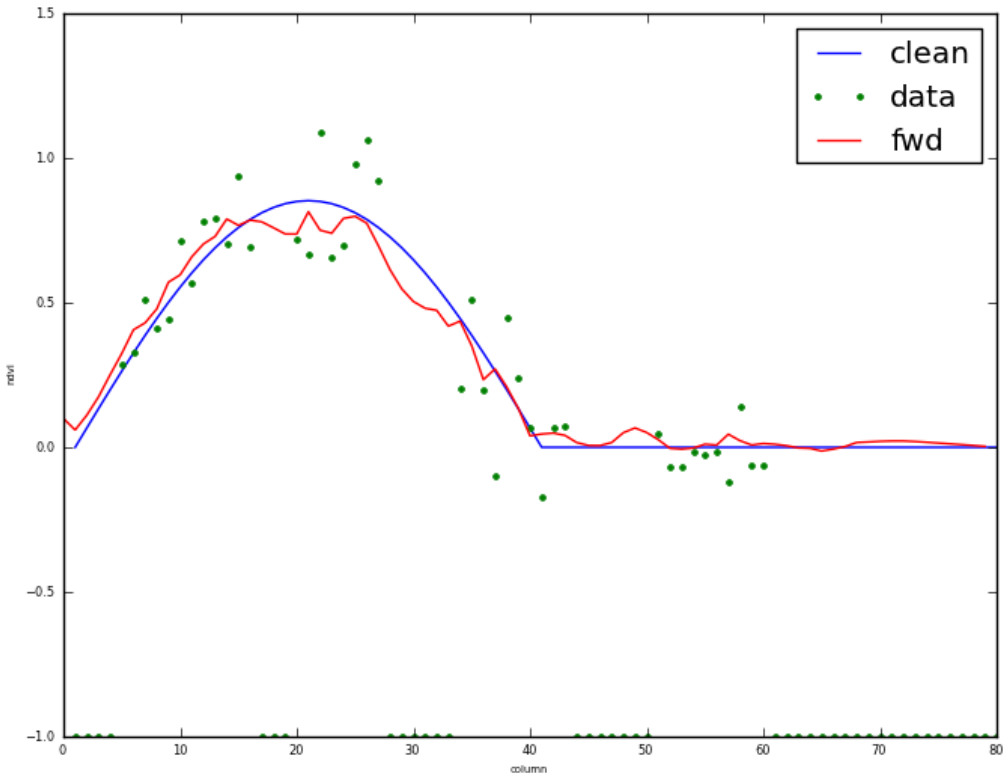


Figure 2.21: Fig. 11: Transect through row 13 of results for 2/3 data missing

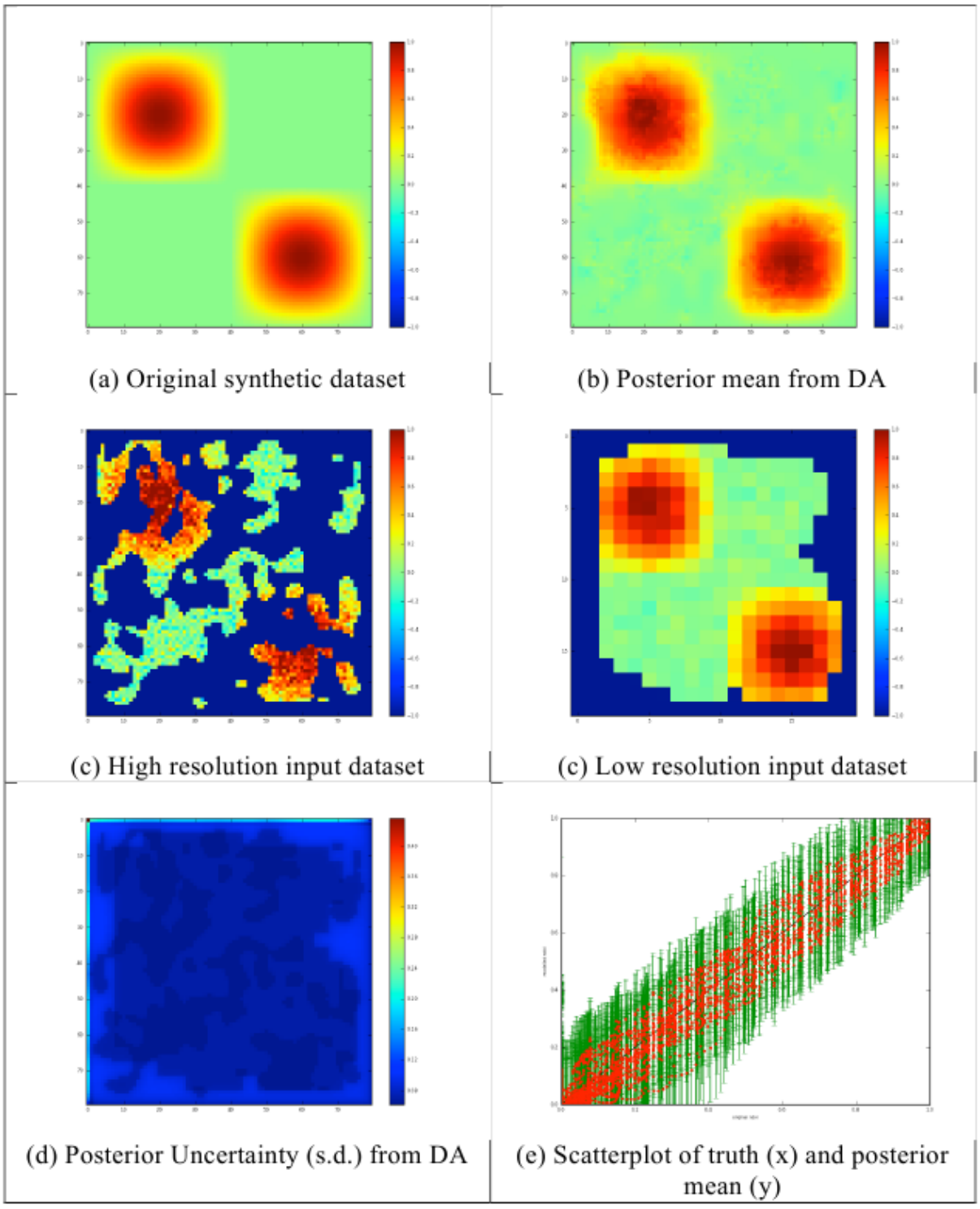


Figure 2.22: Fig. 12: Results of multi-scale analysis for 2/3 data missing in the high resolution and 1/3 missing in the low resolution.

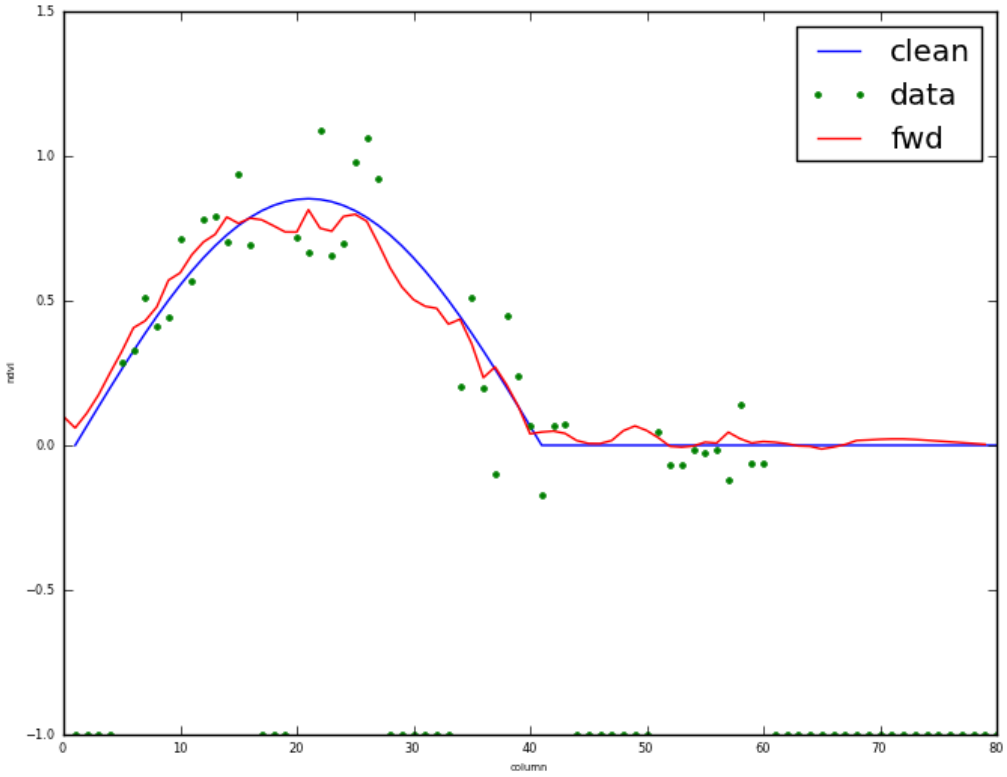


Figure 2.23: Fig. 13: Transect through row 13 of results for 2/3 data missing in the high resolution and 1/3 missing in the low resolution.