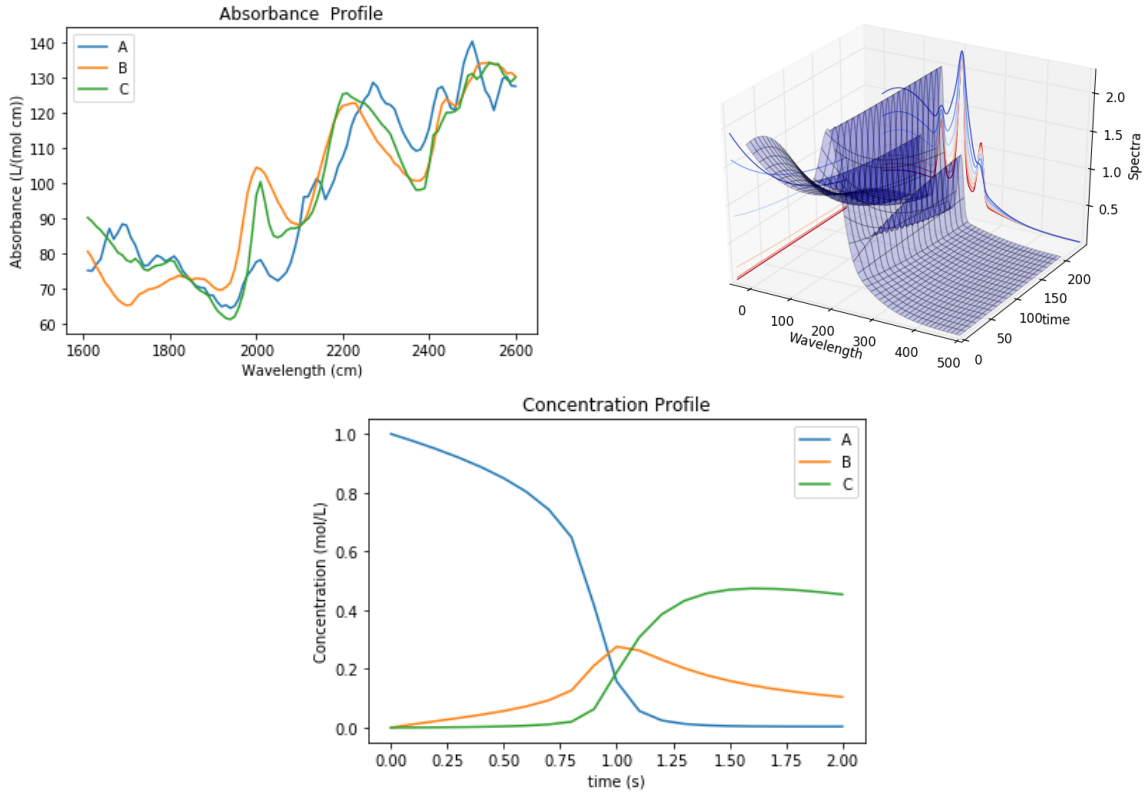
KIPET Documentation

(Kinetic Parameter Estimation Toolkit)

KIPET is an open-source Python package developed through a partnership between Eli Lilly and Company and Carnegie Mellon University. The package is primarily used for the estimation of kinetic parameters from spectral/concentration data. It can also be used to preprocess data, simulate reactive systems, estimate data variances, obtain confidence intervals of the kinetic parameters obtained, and perform estimability analyses. This is all possible through a unified framework based on maximum likelihood principles, robust discretization methods, and large-scale nonlinear optimization.

In this document the capabilities of KIPET are described along with detailed installation instructions and examples and tutorials are provided so that a user with basic programming knowledge can use the toolkit for their own purposes.



For any questions regarding KIPET please contact Prof Larry Biegler: lb01@andrew.cmu.edu

For technical details or to help with the development of the software please join the Github page: https://github.com/salvadorgarciamunoz/KIPET.

KIPET is made available under the GNU General Public License, GPL-3. For more details on this license please review the terms on the Github page or at this link.

KIPET is developed by Salvador Garcia-Munoz (Eli Lilly), Santiago Rodriguez (Purdue University), Lorenz T. Biegler, David M. Thierry, Christina Schenk, and Michael Short (Carnegie Mellon University)

Based on the paper: Chen, W., Biegler, L.T., Garcia-Munoz, S., 2016, An Approach for Simultaneous Estimation of Reaction Kinetics and Curve Resolution from Process and Spectral Data, Journal of Chemometrics, 30, 506-522.

==========

Welcome to KIPET's User Guide!

==========

Contents:

1. OVERVIEW / INTRODUCTION

The Kinetic Parameter Estimation Toolkit (KIPET) is mainly used for solving dynamic parameter estimation problems that arise from chemical reaction systems. The package solves an iterative optimization-based procedure in order to estimate the variances of the noise in system variables (such as concentrations) and spectral measurements. Having estimated the system variances, one can use KIPET to estimate kinetic parameters of chemical reaction models and provide confidence intervals on the estimated parameters. KIPET also contains a host of other useful related tools including estimability analysis, wavelength selection, and data preprocessing. The idea is to provide flexible tools that will allow users to estimate not only kinetic parameters but also the corresponding concentration and absorption profiles (Figures 2) from multi-wavelength spectroscopic data and/or concentration data. (Figure 1)

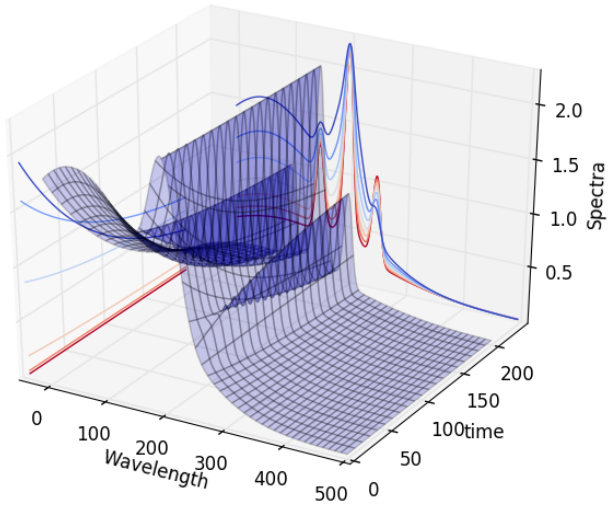


Figure 1: *Visualization of multi-wavelength spectroscopic data*

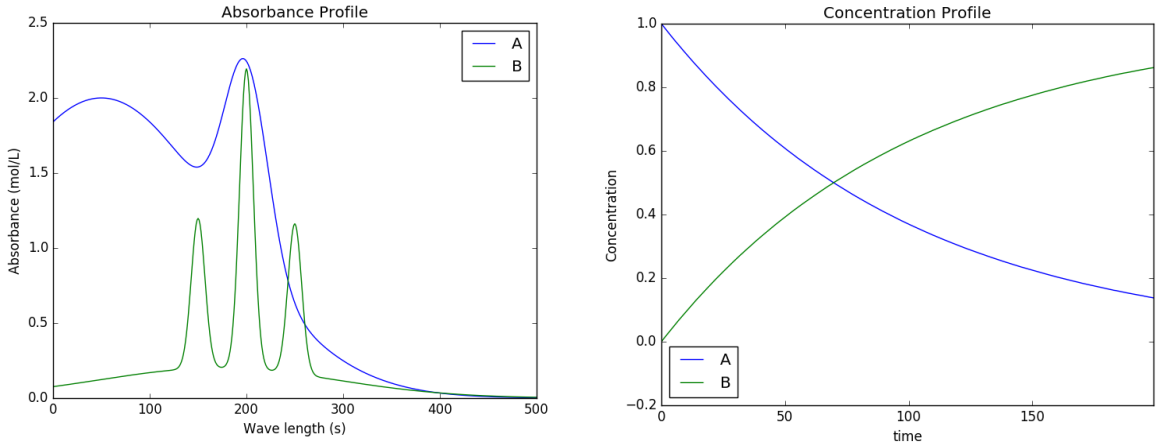


Figure 2*: Absorbance profiles for single components (left) and concentration time profiles (right)*

Additionally, KIPET can be used to obtain a better understanding of a chemical reaction system through inclusion of functionalities that allow for the simulation and optimization of reactive systems that are described by differential algebraic equations (DAEs). The following functionalities are included within the KIPET package:

* Simulation of a reactive system described with DAEs
* Solve DAE systems with collocation methods
* Preprocess data
* Estimability analysis
* Estimate data variances
* Estimate kinetic parameters from single or multiple experimental datasets
* Estimate confidence intervals of the estimated parameters
* Obtain the most informative wavelength set to obtain minimal lack-of-fit
* Tools for system analysis (SVD, PCA, lack of fit, etc.)
* Visualize results

In the sections that follow, this document provides guidelines on how to make use of KIPET. A detailed installation guide is provided, including a list of all packages required to use KIPET. Some background theory into how KIPET works, as well as the structure of KIPET is then explained in Section 3. A guide for using some of the various capabilities is then provided in the form of tutorial examples (Section 4). In Section 5 a more detailed look into some of the classes and functions that are not explicitly explained in Section 4 is provided. Finally, in Section 6, the documentation provides references and further reading that provides more detail on the theory behind the numerical techniques implemented in KIPET.

2. INSTALLATION GUIDE

This section will explain how to install KIPET on different operating systems. Firstly Python will need to be installed.

2.1 Python installation

It should be noted that this guide is intended for beginner users and those that already have Python and an IDE installed can move onto the next section that describes the required packages, their versions, and the KIPET installation instructions.

Check to see whether Python is already installed by going into the command line window and typing “python”. Do this by searching “terminal” in Linux and MacOS, and “cmd” in Windows. If there is a response from a Python interpreter it should include a version number. KIPET should work with any version of Python from 2.7 up until 3.7. If you use Python 3.7 the corresponding tkinter module python3.7-tk needs to be installed as well.

Information on downloading and installing Python for windows is found here. If the user is new to Python and/or uncomfortable with command line interfacing, it is recommended that the user use Anaconda as the Integrated Development Environment (IDE). Anaconda can be downloaded for free and can be installed to include Python, associated packages, as well as a code editor.

Firstly go to the Anaconda download page and select the appropriate option for your operating system and computer architecture.

2.1.1 *Microsoft Windows installation*

For Windows, double-click the .exe downloaded from the site and follow the instructions. During installation the user will be asked to “Add Anaconda to my PATH environment variable” and “Register Anaconda as my default Python X.X”. It is recommended to include both of these options.

After installation, launch Anaconda Navigator by searching for it in your start menu or by launching Anaconda Prompt and typing “anaconda-navigator”.

If you want to use a virtual environment inside Anaconda (useful if you are using Python with different package versions for other purposes), launch the Anaconda Prompt and create one using

```
conda create -n yourenvname python=x.x
```

where yourenvname is the name you choose for your environment and also choose the python version that you got with Anaconda (latest one is 3.7). Then switch to new environment via:

```
source activate yourenvname
```

Then install numpy, scipy, pandas, matplotlib via

```
conda install packagename
```

More info regarding virtual environments in Anaconda, can be found e.g. here.

2.1.2 *MacOS*

For MacOs double-click on the downloaded .pkg installer and follow the prompts. During installation the user will be asked to “Add Anaconda to my PATH environment variable” and “Register Anaconda as my default Python X.X”. It is recommended to include both of these options.

Open Launchpad and click on the Terminal icon. In the Terminal window type “anaconda-navigator”.

2.1.3 *Linux installation*

Enter the following into the terminal to install Anaconda:

bash ~/Downloads/Anaconda3-5.1.0-Linux-x86_64.sh

Follow the instructions and be sure to enter Yes when the installer prompts “Do you wish the installer to prepend the Anaconda 3 install to PATH in your /home/<user>/.bashrc ?”.

Close the terminal in order for the installation to take effect.

To verify the installation, open the terminal and type “anaconda-navigator”. The Anaconda Navigator should open, meaning that you have successfully installed Anaconda.

2.2 Installing Packages and Dependencies

In order to ensure that the dependencies are installed correctly an installer should be used.

2.2.1 Installer

It is recommended to use pip to install all packages and dependencies. If the user already has Python 2.7.9 and up or Python 3.4 and up installed, pip is already included and can be updated using (from the command line or terminal):

```
python -m pip install -U pip
```

This command needs to be performed from the directory where Python is installed. E.g. If you have Anaconda installed you can open navigate to the directory where it is installed and then enter the appropriate commands.

Alternatively you can enter the following in Anaconda Prompt:

```
pip install -U pip
```

For Linux or MacOS If the user is making use of Anaconda it can be updated using:

```
conda update anaconda
```

Note that pip is included in Anaconda and should be up to date if the above is used. If more help is required to install pip, instructions are included here.

2.2.2 Installing Required Dependencies

Now that pip is installed and updated, we can install the other dependencies and third-party packages required for KIPET. Some of the major dependencies include:

* Numpy (van der Walt, 2011): used to support large, multi-dimensional arrays and matrices, http://www.numpy.org/
* Scipy (van der Walt, 2011): used to support efficient routines for numerical integration, http://www.scipy.org/
* Pandas (McKinney, 2013): used to analyze and store time series data, http://pandas.pydata.org/
* Matplotlib (Hunter, 2007): used to produce figures, http://matplotlib.org/
* Pyomo (Hart, 2012): used for formulating the optimization problems http://pyomo.org/

A complete list of all the dependencies is included in Table 1.

If using Anaconda or another scientific Python distribution such as Python(x,y), most of the packages are already installed as part of the distribution. These include Numpy, Scipy, Pandas, and Matplotlib. If using another Python distribution, then each package can be installed individually using pip by the following commands in the MacOS and Linux terminal and the cmd in Windows:

```
python -m pip install -U numpy
```

or via the Anaconda Prompt:

```
pip install -U numpy
```

Where “numpy” is just one example of the packages that will need to be installed.

Please note here, that if you install packages directly, that pyomo needs to be installed as follows:

```
pip install -U pyomo==5.6.1
```

In order to install packages when using Anaconda we can also use the following commands:

```
conda install -c conda-forge pyomo==5.6.1
conda install -c conda-forge pyomo.extras
```

In fact, using Anaconda, Pyomo should be the only additional package to install, as all others should be included in the original environment. It is recommended that the user installs pyomo using the above command in Anaconda prompt in order to ensure pyomo is installed in the correct folder. If any trouble is encountered during installation of any of the dependencies please go to the relevant package websites and follow the more detailed instructions found there.

There is also the possibility of installing all the required dependencies with the adequate versions using:

```
cd kipet
pip install -r requirements.txt
```

When doing this, it is advised to install KIPET first before installing the dependencies. Be aware of that if you need other versions of the required packages for another python-based software, you should rather install KIPET in a virtual environment and run it in this virtual environment. Another thing that should be noted is that if the user is using Windows 7, it is advised to use Python 2.7, rather than Python 3.x and also that there are some known issues with matplotlib in this case. In particular it will be required to install pypng and freetype-py before installing matplotlib. This may therefore cause the requirements.txt to not function correctly.

TABLE 1: *List of dependencies required for KIPET to function*

Package	**Version**
appdirs	1.4.3
backports.functools-lru-cache	1.5
casadi	3.4.0
coverage	4.5.1
cycler	0.10.0
decorator	4.2.1
kiwisolver	1.0.1
matplotlib	2.2.0
networkx	2.1
nose	1.3.7
numpy	1.14.2
pandas	0.22.0
ply	3.11
Pyomo	5.6.1
pyparsing	2.2.0
Python-dateutil	2.7.0
pytz	2018.3
PyUtilib	5.6.5
scipy	1.0.0
six	1.11.0

2.3 Installing KIPET

Firstly, KIPET's source code can be downloaded from https://github.com/salvadorgarciamunoz/kipet.git or through the following command in Linux if git is installed:

```
git clone https://github.com/salvadorgarciamunoz/kipet.git
```

Linux and MacOs

To install KIPET on Linux or MacOS we simply find the directory in the command prompt with the following command:

```
cd kipet
```

and then install using:

```
python setup.py install
```

Microsoft Windows

On Microsoft Windows we can install KIPET by finding either your command prompt or Anaconda Prompt and going into the KIPET folder using:

```
cd kipet
```

And then using:

```
python setup.py install
```

2.4 Installing solver / IPOPT

Currently the only nonlinear solver implemented and tested in KIPET is IPOPT (Wächter and Biegler, 2006). This document only provides basic instructions on the easiest method to install the solvers. For a detailed installation guide please refer to the COIN-OR project website. If you have purchased or obtained access to the HSL solver library for additional linear solvers, the instructions for this compilation are also found on the COIN-OR website.

Linux/MacOS installation

Download the IPOPT tarball and then issue the following commands in the relevant directory:

```
gunzip Ipopt-x.y.z.tgz
tar xvf Ipopt-x.y.z.tar
```

Where the version number is x.y.z. Rename the directory that was just extracted:

```
mv Ipopt-x.y.z CoinIpopt
```

Then go into the directory we just created:

```
cd CoinIpopt
```

and we create a directory to move the compiled version of IPOPT to, e.g.:

```
mkdir build
```

and enter this directory:

```
cd build
```

Then we run the configure script:

```
../configure
```

make the code

```
make
```

and then we test to verify that the compilation was successfully completed by entering:

```
make test
```

Finally we install IPOPT:

```
make install
```

Microsoft Windows

The simplest installation for Microsoft windows is to download the pre-compiled binaries for IPOPT from COIN-OR. After downloading the file and unzipping it you can place this folder into the Pyomo solver location:

```
C:\Users\USERNAME\Anaconda3\Lib\site-packages\pyomo\solvers\plugins\solvers
```

Run an example (explained in the next section) to test if it works. This method should also include a functioning version of sIpopt and so the next step is not necessary unless another method of installation is used.

If trouble is experienced using this approach other methods can be used and they are detailed in the *Introduction to IPOPT* document.

Another simple way to install IPOPT for use in the Anaconda environment is to use the following within the Anaconda Prompt:

```
conda install -c conda-forge ipopt
```

Note that this version of IPOPT is not necessarily the most up-to-date and will not have access to the more advanced linear solvers that are available through the HSL library, and so it is rather advised to compile the solver for your own use.

2.5 Installing k_aug

If the user would like to utilize k_aug to perform confidence intervals or to compute sensitivities, k_aug needs to be installed and added to the system path. A complete guide can be found within the same folder as this documentation on the Github page, or can be found in David M. Thierry's Github page https://github/davidmthierry/k_aug . David has also kindly produced a Youtube video that shows how to install k_aug on Windows.

k_aug is a necessary component if the user would like to make use of the estimability analyses offered within KIPET.

sIPOPT installation

We have decided to remove sIPOPT from most of the KIPET package as the software has not been maintained for over 10 years and because k_aug has more flexibility. If, however, for whatever reason, you would like to use sIPOPT, it can still be used sensitivity analysis.

Additional information on how to install sIpopt can be found in:

https://projects.coin-or.org/Ipopt/wiki/sIpopt

It is important to notice here that the instructions for Windows, if the solver was installed as shown above, will not work with sIpopt as no binaries for sIpopt are available. Because of this you will need to follow the Cygwin installation instructions provided by the IDAES group's Akula Paul, included in the same folder as this documentation with the filename of: "Ipopt_sIpopt_Installation_on_Windows_cygwin.pdf".

2.6 Windows PATH Management

If there are issues found with running examples it may be necessary in Windows to add Python to the PATH environmental variable. This can be done through your IDE, Spyder, in the case of this document by following these steps. Navigate to to Tools>PYTHONPATH Manager in Spyder and add the folder C:\Users\Username\Anaconda3 to the PATH.

If the user would like to use Python commands from the command prompt, as opposed to the Anaconda prompt, then Python can be added to the PATH Environmental Variable by going into the Start Menu, right-clicking on *My Computer* and clicking on *Advanced Settings* in *Properties*. In this window one can find "Environment Variables". Click Edit and add Python to the PATH variable by adding the location of where Python is installed on your system.

You should now be ready to use KIPET!

2.7 Validation of the Package

To test that the package works, there is a test script provided that checks all of the functions within KIPET through the running of multiple examples in series. The examples can take a while to run. If some of the tests do fail then it is possible something is wrong with the installation and some debugging may need to take place. To run the validation script go into the KIPET folder, enter the validation folder and run 'validate_installation.py'.

python validate_installation.py

Note that if sIpopt or k_aug are not installed, certain test problems will fail to run. If this is the case and you do not intend to use the sensitivity calculations, then ignore these failures.

2.8 Updating KIPET

Repeat steps 2.2., 2.3 and 2.7 with the new version downloaded from github. This is now your new work directory.

3. BACKGROUND

This documentation focuses on kinetic studies for the investigation of chemical reactions and identification of associated rate constants from spectroscopic data. The methodology is the same as published in Chen, et al. (2016), where the technical details are laid out in significant detail. In this document the user will find a summary of the procedure in the paper as well as how this method has been transferred to KIPET. This document will therefore attempt to only describe as much detail as necessary in order to understand and use KIPET.

3.1 General modeling strategy and method

After installing and importing the package users can do the following things:

* Build a chemical reaction model
* Simulate the model
* Estimate variances in the data
* Preprocess data
* Perform estimability analysis
* Estimate parameters
* Ascertain whether a different subset of wavelengths is more suitable for the model
* Compute confidence intervals of the estimated parameters
* Plot concentration and absorbance profiles

The first step in KIPET is to create a model. A model contains the physical equations that represent the chemical reaction system dynamics. Once a model is created users can either make a simulation by solving the DAE equations with a multi-step integrator or through a discretization in finite elements. Alternatively an optimization can be performed in which the DAE equations are the constraints of the optimization problem. In general, KIPET provides functionality to solve optimization problems for parameter estimation of kinetic systems. For the construction of optimization models KIPET relies on the Python-based open-source software Pyomo. Pyomo can be used to formulate general optimization problems in Python. After a model is created users can extend the model by adding variables, equations, constraints or customized objective functions in a similar way to Pyomo. After the simulation or the optimization is solved, users can visualize the concentration and absorbance profiles of the reactive system. These steps are summarized in the following figure (Figure 3):

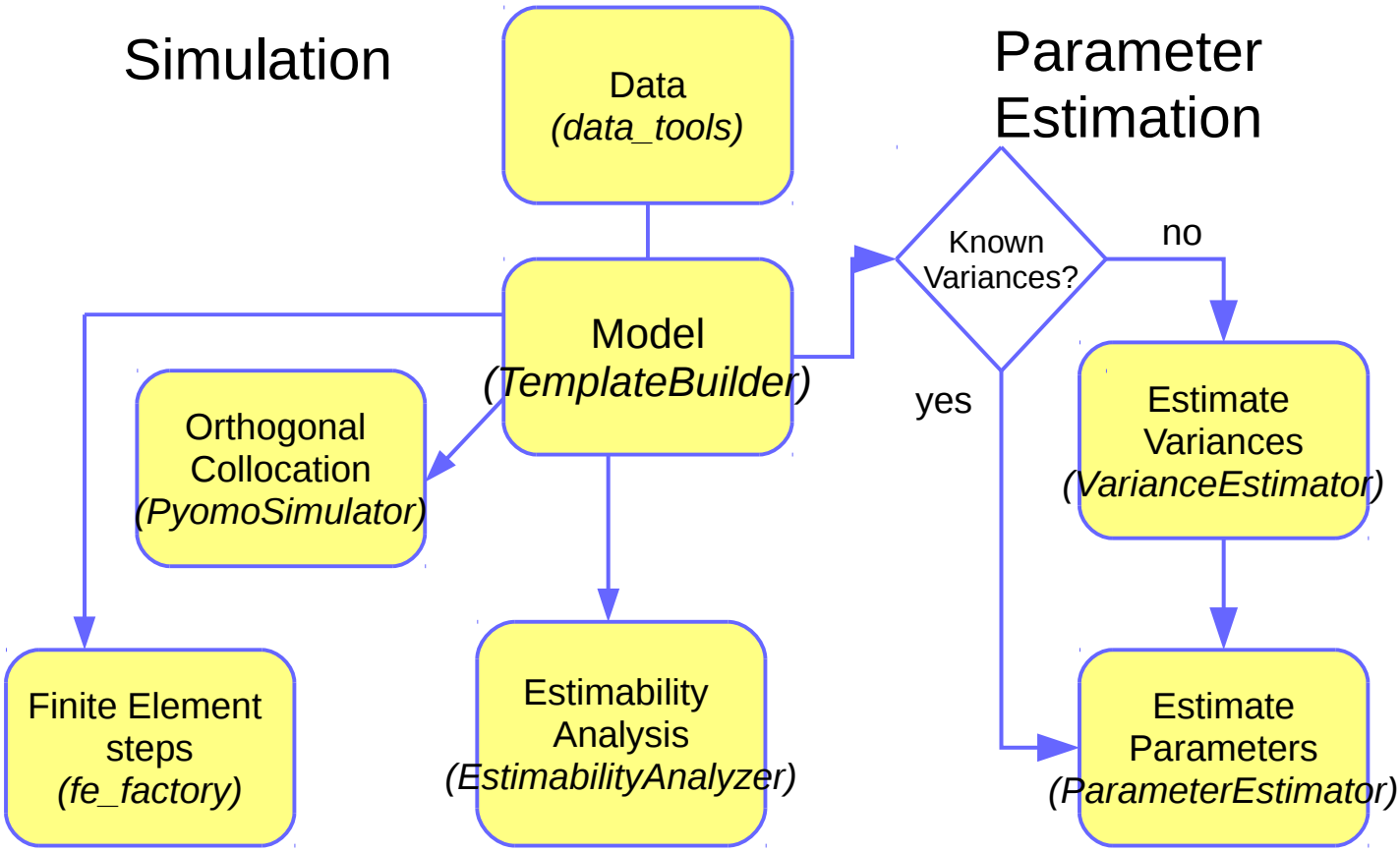


Figure 3: *The steps/modules involved in setting up a KIPET model*

It should be noted that in the above figure, the location of the source-code for each module is included in the orange text in the top-left section of each box. You may wish to refer back to this figure during the setting up of models to decide on which modules from within KIPET to use and call upon.

To facilitate the creation of models KIPET provides a helper class that makes the creation of kinetic models easier. The helper class TemplateBuilder transforms the user's model into a Pyomo formulation "under the hood" automatically. In general the TemplateBuilder creates models with the following attributes:

* Noised concentration variable of each component, C
* Unnoised/actual concentration variable of each component, Z
* Pure component absorption variable, S
* Complementary state variables, X
* Complementary algebraic variables, Y
* Kinetic parameters, P

The variable nomenclature follows the same labeling structure as used in the original paper, Chen et al. (2016). Once the model is created it can be simulated or optimized. KIPET simulates and optimizes Pyomo models following a simultaneous approach. In the simultaneous approach all of the time-dependent variables are discretized to construct a large nonlinear problem. Due to the nature of large nonlinear problems, good initial guesses for variables are essential. KIPET provides a number of tools to help users to initialize their problems, including through the use of running simulations with fixed guessed parameters, using a least squares approach with fixed parameters, or through a finite element by finite element approach using KIPET's in-built Fe_Factory (recommended for large problems).

KIPET therefore offers a number of simulator and optimizer classes that facilitate the initialization and scaling of models before these are called for simulation. In addition, the simulator and optimizer classes available in KIPET will store the results of the simulation/optimization in pandas DataFrames for easy visualization and analysis. More information on this and why this is relevant to the user will follow during the tutorial problems.

KIPET offers two classes for the optimization of reactive models. The ParameterEstimator class estimates kinetic parameters from spectral data by solving the problem formulation described in Chen, et al. (2016). Within this class the objective function is constructed with Pyomo and added to the model that is passed to the solver. If the user provides a model with an active objective function however, the ParameterEstimator will optimize the objective function provided by the user. This class also offers the ability to determine the confidence intervals of the estimated parameters. For all calculations in the ParameterEstimator class the variances of the spectral data need to be provided. When the variances are not known the user can use the VarianceEstimator optimizer class instead to determine them.

The variance estimation procedure is described in detail in Chen, et al. (2016). The procedure consists of solving three different nonlinear optimization problems in a loop until convergence on the concentration profiles is achieved. The following figure summarizes the variance estimation procedure based on maximum likelihood principles:

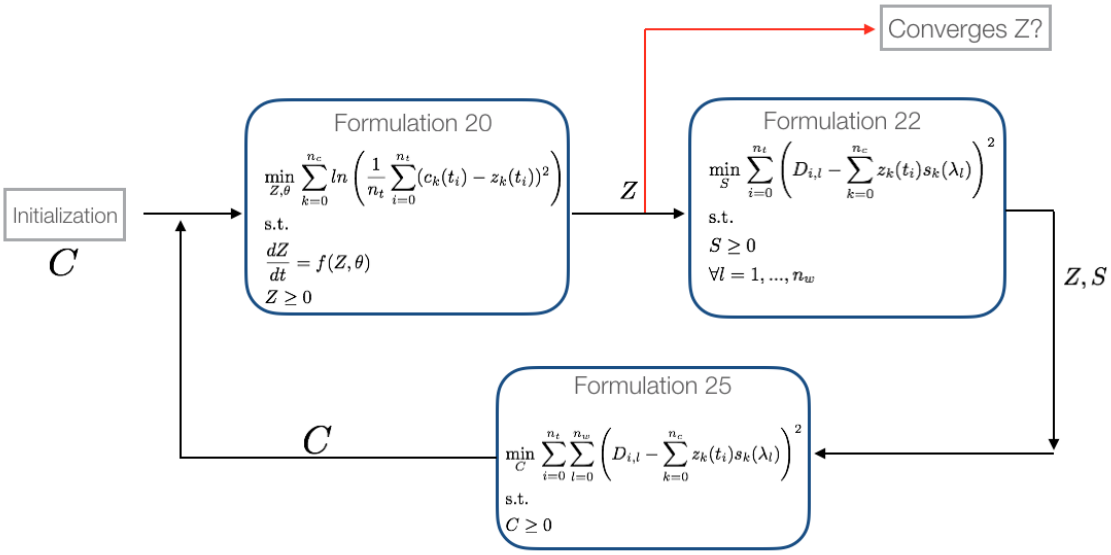


Figure 4: *The VarianceEstimator class algorithm*

The VarianceEstimator class will construct the three problems and solve them with a nonlinear solver until the convergence criteria is satisfied. By default KIPET checks for the infinite norm of the residuals of Z between two iterations. If the infinity norm is less than the tolerance (default 5e-5) then variances are estimated by solving the overdetermined system shown in Figure 5.

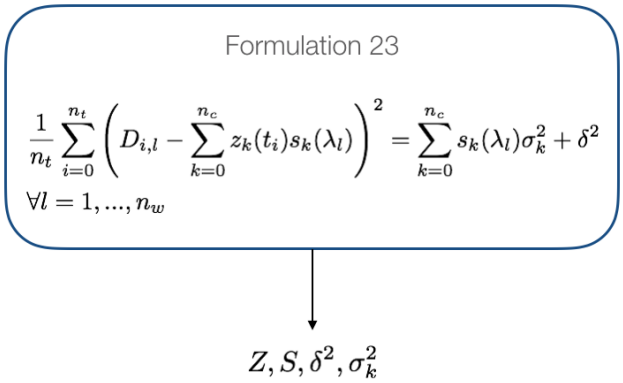


Figure 5: *Variance estimation equations*

The solution of each subproblem in this procedure is logged in the file iterations.log. Examples on how to use the optimization classes and their corresponding options can be found in the tutorial section of this document. It should be noted at this point that all that is required to determine the variances in this way are the components, their initial concentrations, the differential equations describing the reactions, and the spectroscopic data absorption matrix, D, which consists of the experimental absorption data with rows (i) being the time points and columns (l) being the measured wavelengths.

Once the variances are estimated we not only attain good estimates for the system noise and the measurement error, but we have also obtained excellent initializations for the parameter estimation problem, as well as good initial values for the kinetic parameters to pass onto the ParameterEstimator class.

Where Equation 17 from Chen, et al. (2016) is solved directly:

$$\begin{aligned}
&\min \sum_{i=1}^{ntp}\sum_{l=1}^{nwp}\left(d_{i,l}-\sum_{k=1}^{nc} c_k(t_i)s_k(\lambda_l)\right)^2/\delta^2+\sum_{i=1}^{ntp}\sum_{k=1}^{nc}\left(c_k(t_i)-z_k(t_i)\right)^2/\sigma_k^2 \\
&\text{s.t.}\quad \sum_{m=0}^{K}\dot{l}_m(\tau)z_{jm}-h_j\boldsymbol{f}\left(z_{jm},\boldsymbol{\theta}\right)=0, j=1..ne, m=1..K \\
&\qquad z^K(t_i)=\sum_{m=0}^{K} l_m(\tau)z_{jm}, \tau=\left(t_i-tp_{j-1}\right)/\left(tp_j-tp_{j-1}\right) \\
&\qquad \boldsymbol{C}\geq 0, \boldsymbol{S}\geq 0
\end{aligned} \tag{17}$$

Note here that this can be solved either directly with the variances and measurement errors manually added and fixed by the user, or through the use of the VarianceEstimator.

It is also important at this point to note that we can solve the ParameterEstimator problem either using IPOPT to get the kinetic parameters or we can use sIPOPT to perform the optimization with sensitivities in order to determine the confidence intervals.

3.2 Setting up and understanding a model in KIPET

Once all of the software required for KIPET is installed, we can begin learning and testing the toolkit by opening the "Ex_1_ode_sim.py". The template examples are some simple tutorial examples designed to assist new users in utilizing the software and are also meant to allow for easy manipulation so that the user can alter the code for their own specific uses.

Before getting started with KIPET it is useful for users to be familiar with basic Python programming. The Python website provides users with a basic tutorial.

If you are new to Python or a novice in coding, you can open the example by starting the IDE of your choice and loading the example from the folder where KIPET is installed. For those using the recommended Anaconda platform, launch Anaconda-Navigator (either from the start menu in Windows or by typing "anaconda-navigator" into the terminal in Linux/MacOS) and then launching the Spyder IDE from the Homepage.

In this section the user will be guided through the various types of problems that KIPET was designed to solve, with detailed line-by-line analysis of the example files, with particular attention being given to the areas of the code where it is intended for users to interact in order to implement their own problems. The tutorials section will be broken down into:

General model details

This is information that needs to be supplied by the user for every problem to be solved in KIPET.

Data handling

In this section we will look at which types of data can be handled by KIPET and what form the data files need to be supplied in.

Simulation

This section deals with the simplest type of problem that can be solved in KIPET, involving plotting concentration vs time graphs for the specified systems with the systems' parameters specified. If the individual pure species' absorbances (S-matrices) are known these can also be inputted to obtain the expected absorbance profiles (D-matrix) for the system.

Variance Estimation

This section will show how to use the VarianceEstimator class to estimate the measurement error and noise in the data.

Parameter Estimation

KIPET's most valuable and powerful class is described here, allowing for the estimation of kinetic parameters. This section will explain how the user calls upon the ParameterEstimator class and how to solve the discretization and optimization problem.

Advanced problems with complementary states

This section will demonstrate the way in which more advanced systems can be solved by introducing complementary state variables and algebraic variables to introduce complexities that may also change with time such as temperature, densities, volumes, etc.

4. TUTORIALS

This section will look at specific example files provided with the KIPET software and explain, line-by-line the sections that a user will need to modify in order to solve their own problems. All lines of code that are not explained in this tutorial are necessary for the code to function, but should not need to be altered by the user. For brevity these lines are omitted.

Firstly we will look at the pieces of information that every KIPET model requires to function.

4.1 General Model Details

In order to fully understand the model-building process and results obtained, the user needs to be familiar with the nomenclature used.

Variables:

Z -- unnoised concentrations of each species, continuous in time.

C – concentration matrices with noise.

S -- pure component absorbance matrices.

X – algebraic variables such as volume, temperatures, densities, etc.

P – kinetic parameters (can be fixed for simulations)

Parameters and (indices):

Components, (k) – the components, with index k

Meas_times, (t) – the measurement times from the data file.

Meas_lambdas, (l) – the wavelengths from the experimental data.

Every KIPET model requires a number of things in order to function. This includes a list of the components in the system, identification of the components that will be absorbing, a list of the reactions (noting that each component requires an ODE), and a number of class and function calls that will be detailed in the sections to follow.

Generally the code can be broken down into a number of steps, with some of the steps only being applicable to certain model applications. Each of these steps will be described in the sections to follow within this document, with examples as to how to use the relevant functions for your own problems. Please note that, at the moment of writing this document, it is required that you import the library that you are using in each example. The import statement for each of the sets of tools described below is included. These steps are summarized as:

1. Reading / manipulating / preprocessing data

Required for problems where data is available or needs to be generated. A wide variety of reading, writing and data manipulation tools are available within this module.

```
from kipet.library.data_tools import *
```

2. TemplateBuilder

This section is required for all models and is where the components, ODEs, and other problem-specific data is inputted.

```
from kipet.library.TemplateBuilder import *
```

3. Simulator

This section is required if a simulation is being performed and can call either upon PyomoSimulator of FESimulator. Noise can also be simulated and added to the results or used to generate a D-matrix. The simulators can also be used to initialize the parameter estimation problem if needed.

```
from kipet.library.PyomoSimulator import *
```

If the fe_factory is used to initialize the simulator then we can use the FESimulator module, which acts as an automatic wrapper function to use fe_factory with KIPET models.

```
from kipet.library.FESimulator import *
from kipet.library.fe_factory import *
```

4. VarianceEstimator

This is meant for problems where we have a D-matrix from experimental data with measurement error and random noise. Can also be used to initialize the kinetic parameter estimation.

```
from kipet.library.VarianceEstimator import *
```

5. ParameterEstimator

Used to simultaneously determine the concentration-time profiles and kinetic parameters using orthogonal collocation on finite elements, as described previously in this document. Can also determine the covariances and confidence intervals associated with the kinetic parameters. Module also contains the tools used to determine the appropriate subset of wavelengths to be used for the parameter estimation based on a lack-of-fit.

```
from kipet.library.ParameterEstimator import *
```

6. EstimabilityAnalyzer

Uses k_aug to obtain sensitivities and then uses these sensitivities to obtain the parameter estimability rankings. These ranked parameters are then used to compute mean squared errors for various simplified models using some paramters fixed and others remaining as variables. Currently only functional for concentration-data problems.

```
from kipet.library.EstimabilityAnalyzer import *
```

7. Visualising and viewing results

Finally we can visualize the results by using the matplotlib graphing functions as well as printing the variances, kinetic parameters, and confidence intervals.

The next sections will provide some tutorials as to allow the user to use all of KIPET's functionality.

4.2 Tutorial 1 – Simulating a Simple Example

If you do not know how to get started, open Anaconda Navigator and then run Spyder IDE. In Spyder open the example by navigating into KIPET's example folder and opening the relevant example. Run the example by pressing the green play button. We will start by going through the example "Ex_1_ode_sim.py".

This example provides a basic 3-component, 2 reaction system with A → B and B → C, where the kinetic rate constants are fixed.

4.2.1 TemplateBuilder

The first line of interest to the user will be the line:

```
builder = TemplateBuilder()
```

Where we use the required TemplateBuilder class to begin creating our model. This class is required for every KIPET example and allows the user to construct the Pyomo model in the background. While the user does not need to interact with this line, it is useful to realise its importance. The lines that follow allow us to define the components that we expect to be present in the mixture.

```
builder.add_mixture_component('A',1)
builder.add_mixture_component('B',0)
builder.add_mixture_component('C',0)
```

Where the first function input is the name of the component and the second input is the initial concentration in the mixture. Following this is the next required user input, the definition of the kinetic parameters:

```
builder.add_parameter('k1',2.0)
builder.add_parameter('k2',0.2)
```

Where the first function input is the name of the kinetic parameter and the second input is the value. For this first example we are fixing the values of the kinetic parameters in order to simulate the system. If we wish to optimize the system we can either leave this second argument out or add bounds, like in the following example:

```
builder.add_parameter('k1', bounds=(0.0, 5.0))
builder.add_parameter('k2', bounds=(0.0, 1.0))
```

Following the definition of our basic reaction system, every KIPET model requires a set of ODEs to be defined. Please note that KIPET requires that every component that is identified in the mixture has an expression. The ODEs are inputted in the following way for this example:

```
def rule_odes(m,t):
      exprs = dict()
      exprs['A'] = -m.P['k1']*m.Z[t,'A']
      exprs['B'] = m.P['k1']*m.Z[t,'A']-m.P['k2']*m.Z[t,'B']
      exprs['C'] = m.P['k2']*m.Z[t,'B']
      return exprs
```

After defining the equations in this way we can add them to our template which will create the Pyomo model using:

```
builder.set_odes_rule(rule_odes)
```

So now that the model is defined we can decide what to do with it. In this example the goal is to simulate the known reaction system and obtain concentration vs time profiles.

4.2.2 PyomoSimulator

We will do this by sending our “builder” model through the PyomoSimulator class, which discretizes the system and solves an optimization problem using orthogonal collocation on finite elements. Firstly, we define our pyomo model using our builder TemplateBuilder:

```
pyomo_model = builder.create_pyomo_model(0.0,10.0)
```

Where the two arguments are the time period that we would like to simulate over. We then pass this new model onto the PyomoSimulator class:

```
simulator = PyomoSimulator(pyomo_model)
```

and apply the discretization that we would like for the ODEs:

```
simulator.apply_discretization('dae.collocation',nfe=60,ncp=3,scheme='LAGRANGE
                                    -RADAU')
```

The arguments that need to be supplied use the same keywords as the pyomo.dae method. And need to include all the arguments above. Where “nfe” is the number of finite elements, the higher the number the more accurate your solution is likely to be but at the cost of higher computational costs. “ncp” is the number of collocation points within each finite element. As this number increases, the computational costs increase dramatically and the chances of a converged/feasible solution decrease. The “scheme” refers to the type of collocation to be applied, with two options available to users, either collocation using Lagrange-Radau (as above) roots or using Lagrange-Legendre (scheme=’LAGRANGE-LEGENDRE’). It is highly recommended that the user use Lagrange-Radau (which is also the default option if the argument is omitted) as this has been shown to produce more numerical stability for complex problems, such as the ones usually encountered in KIPET.

Finally we are ready to run the simulation in Pyomo with the following line:

```
results_pyomo = simulator.run_sim('ipopt',tee=True)
```

where ‘ipopt’ is the IPOPT nonlinear program (NLP) optimization solver and the option “tee” is to stream the solver output to the console. For more information on additional arguments, please refer to the function guide in section 5.2.

4.2.3 Visualizing and viewing results

Finally, to view the results of the optimization-based simulation we use the matlplotlib function

```
if with_plots:
    results_pyomo.Z.plot.line(legend=True)
    plt.xlabel("time (s)")
    plt.ylabel("Concentration (mol/L)")
    plt.title("Concentration Profile")
    plt.show()
```

Where we are plotting the concentration vs time graph with the unnoised concentration, Z. The plot obtained from this example is shown in Figure 6.

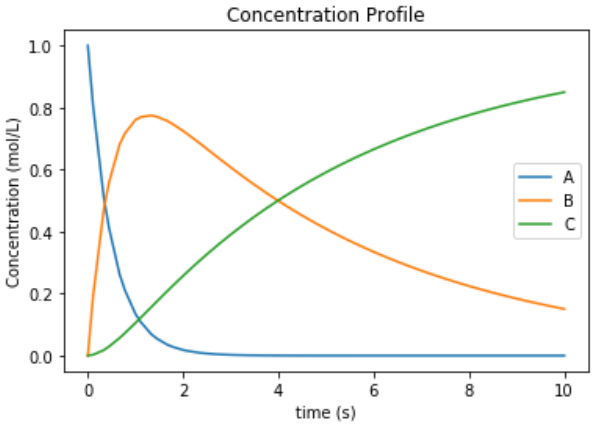


Figure 6: *Plot obtained from tutorial example 1*

4.3 Tutorial 2 – Parameter Estimation

In the second example we will be looking at a parameter estimation problem where we combine most of the elements discussed in the Overview section of this document. This model is label “Ex_2_estimation.py”. This example is the same reaction system as tutorial 1, except in this case we use a simulated data set as our input D-matrix. This example, while not too computationally complex provides a good overview of all the capabilities of KIPET.

4.3.1. Reading data

Firstly we will need to input our D-matrix, which contains the spectral data. More notes on the structure of these data files and which file types are permitted are included in the section on data files. In order to do this we need to point the model to the data file. We can do this by using the following lines:

```
dataDirectory = os.path.abspath(os.path.join( os.path.dirname(
                os.path.abspath(inspect.getfile(
                  inspect.currentframe() ) ) ), '..','data_sets'))
filename =  os.path.join(dataDirectory,'Dij.txt')
```

```
D_frame = read_spectral_data_from_txt(filename)
```

where dataDirectory points to the folder/directory where the data file is found. The '..' refers to changing directory back one and 'data_sets' is the name of the folder where the data file is found. The filename refers to the exact location of the data file, 'Dij.txt'. Finally we define the D_frame which uses the KIPET function read_spectral_data_from_text().

4.3.2. TemplateBuilder

The TemplateBuilder is constructed in the same way as in the previous example, except in this case we add the parameters with bounds as opposed to fixed values. This is done in the following way:

```
builder.add_spectral_data(D_frame)
```

The components, ODEs, and the optimization problem are all set up in the same way as was previously explained.

4.3.3. VarianceEstimator

After the Pyomo model is set up with the TemplateBuilder the VarianceEstimator is called, followed by the discretization scheme, as was previously shown:

```
v_estimator = VarianceEstimator(opt_model)
 v_estimator.apply_discretization('dae.collocation',nfe=60,ncp=1,scheme='LAG
RANGE-RADAU')
```

The next section of code run the optimization procedure described in the Overview section and detailed in Chen, et al. (2016).

```
options = {}
A_set = [l for i,l in enumerate(opt_model.meas_lambdas) if (i % 4 == 0)]
results_variances = v_estimator.run_opt('ipopt', tee=True,
                                        solver_options=options,
                                        tolerance=1e-5,max_iter=15,
                                        subset_lambdas=A_set)
```

Where the solver is 'ipopt', tee = True is to stream the optimizer output to the console, solver options are related to the IPOPT solver and can be viewed in the IPOPT manual. In the case of this example, no options are given to the solver and allowing KIPET to provide good default options. The tolerance argument is the tolerance required for the termination of the variance estimator for the change in Z between iterations, as described in the paper. The max_iter argument is for the maximum number of iterations of the iterative procedure described in Figure 4.

subset_lambdas = A_set is the set of wavelengths to use in initializing the problem, default=all. For large problems it might be worth making that smaller as it allows the VarianceEstimator a smaller set to work with. For problems with a lot of noise, this can be very useful and was shown in the paper to be equally effective as including the entire set of wavelengths. A_set in this example is set to be every fourth value.

After the VarianceEstimator iterative procedure is completed, the results can then be printed:

```
print "\nThe estimated variances are:\n"
for k,v in results_variances.sigma_sq.items():
```

```
    print k,v
sigmas = results_variances.sigma_sq
```

This should output the following for this example:

```
The estimated variances are:

A 4.5576011654506495e-12
device 1.8751166756864294e-06
C 2.412124777715806e-11
B 3.33664956457665e-11
```

Where the “device” refers to the measurement error.

4.3.4. ParameterEstimator

After rewriting the Pyomo model created from the VarianceEstimator, the ParameterEstimator function is created:

```
p_estimator = ParameterEstimator(opt_model)
```

Discretization is then applied to the p_estimator, as in the previous example, and before running the ParameterEstimator optimization we can use the VarianceEstimator’s results in order to initialize the p_estimator model. This is only possible if the VarianceEstimator is used, but it can also be omitted. Note that the discretization of the p_estimator has to be done with the same element and collocation point placement in order to maximize the benefits of the initialization step.

```
p_estimator.initialize_from_trajectory('Z',results_variances.Z)
p_estimator.initialize_from_trajectory('S',results_variances.S)
p_estimator.initialize_from_trajectory('C',results_variances.C)
```

IPOPT performs rudimentary scaling of variables automatically. Some problems may, however, require more detailed scaling information so that IPOPT can perform in an efficient manner. In order to use the scaling information from the previous optimization we may use the following:

```
p_estimator.scale_variables_from_trajectory('Z',results_variances.Z)
p_estimator.scale_variables_from_trajectory('S',results_variances.S)
p_estimator.scale_variables_from_trajectory('C',results_variances.C)
```

Once again, this step is not a necessity for all problems, particularly those that are well-scaled for IPOPT. If this variable scaling is included then the optimization step will need to include the NLP scaling option, as demonstrated below:

```
options = dict()
options['nlp_scaling_method'] = 'user-scaling
results_pyomo = p_estimator.run_opt('ipopt', tee=True, solver_opts =
  options, variances=sigmas)
```

Where the additional argument, variances = sigmas, refers to the fact that we are including the variances calculated by the VarianceEstimator.

4.3.5. Confidence intervals

If the user would like to assess the level of confidence in the estimated parameters the run_opt function needs to be changed. And example of this is found in the Example labeled “Ex_2_estimation_conf.py” in the Examples folder. Firstly the ‘k_aug’ solver needs to be called. Additionally, the option for the covariance needs to be changed from the default. More information on the ParameterEstimator function is found in section 5.

```
options = dict()
options['mu-init'] = 1e-4
results_pyomo = p_estimator.run_opt('k_aug',  tee=True, solver_opts =
                                options, variances = sigmas,
                                 covariance= True)
```

Please note that if this fails to run, it is likely that sIPOPT is not correctly installed, or it has not been added to your environmental variable. For help with sIPOPT, please refer to section 2.4.

For many of the problems it is not possible to use the user scaling option as the solver type has now changed. In addition, since the stochastic solver requires the solving of a more difficult problem, it is sometimes necessary to apply different solver options in order to find a feasible solution. Among the options commonly found to increase the chances of a feasible solution, the ‘mu-init’, option can be set to a suitably small, positive value. This option changes the initial variable value of the barrier variable. More information can be found on the IPOPT options website in COIN-OR.

4.3.6. Visualizing and Viewing Results

Once the optimization is successfully completed we can print the results:

```
print "The estimated parameters are:"
for k,v in results_pyomo.P.items():
    print k,v
```

The results will then be shown as:

```
EXIT: Optimal Solution Found.
The estimated parameters are:
k2 0.201735984306
k1 2.03870135529
```

Which will be the estimates for parameters k1 and k2. Finally we use the same methods to display results as in the first example, but now also displaying the plots for the S (pure component absorbance) matrices:

```
results_pyomo.C.plot.line(legend=True)
plt.xlabel("time (s)")
plt.ylabel("Concentration (mol/L)")
plt.title("Concentration Profile")
results_pyomo.S.plot.line(legend=True)
plt.xlabel("Wavelength (cm)")
plt.ylabel("Absorbance (L/(mol cm))")
```

Providing us with the following plots:

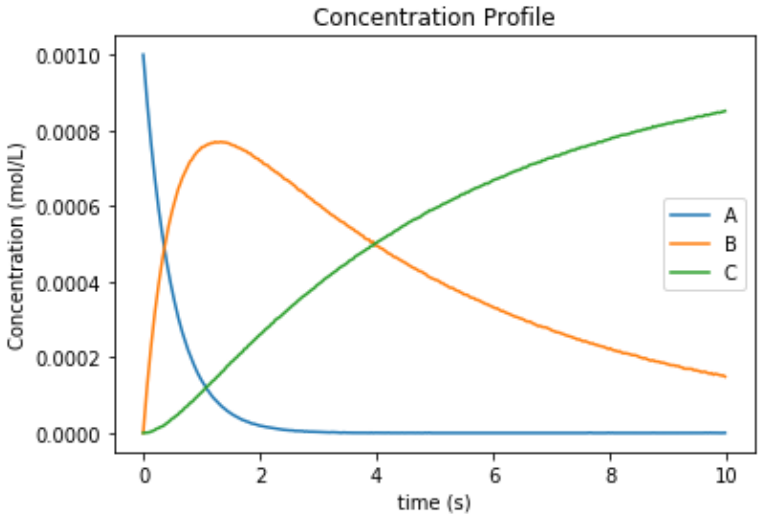


Figure 7: *Concentration profile results from tutorial example 2*

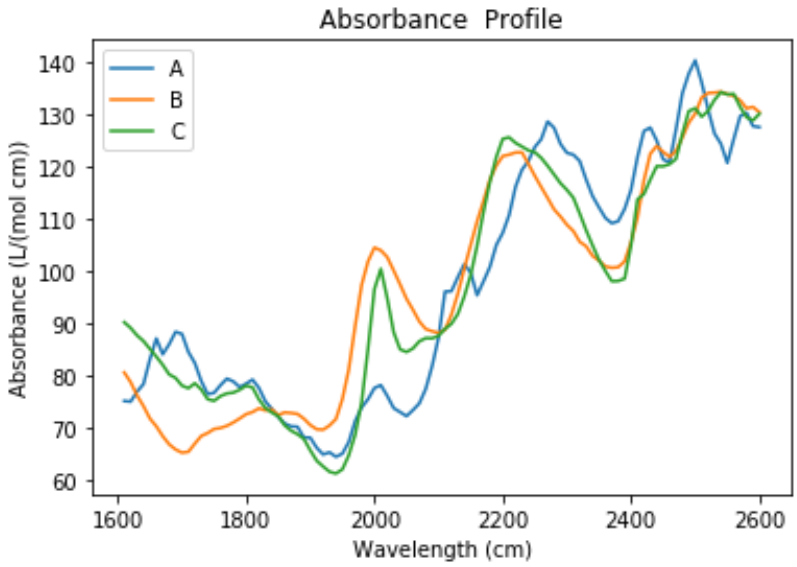


Figure 8: *Pure component absorbance profiles (S) result from tutorial example 2*

4.4 Tutorial 3 – Advanced reaction systems with additional states

It is also possible to combine additional complementary states, equations and variables into a KIPET model. In the example labeled “Ex_3_complementary.py” an example is solved that includes a temperature and volume change. In this example the model is defined in the same way as was shown before, however this time complementary state variable temperature is added using the following KIPET function:

```
builder.add_complementary_state_variable('T',290.0)
builder.add_complementary_state_variable('V',100.0)
```

This function now adds additional variables to the model, labeled “X”. This same formulation can be used to add any sort of additional complementary state information to the model. Now, similarly to with the components, each complementary state will require an ODE to accompany it. In the case of this tutorial example the following ODEs are defined:

```
def rule_odes(m,t):
    k1 = 1.25*exp((9500/1.987)*(1/320.0-1/m.X[t,'T']))
    k2 = 0.08*exp((7000/1.987)*(1/290.0-1/m.X[t,'T']))
    ra = -k1*m.Z[t,'A']
    rb = 0.5*k1*m.Z[t,'A']-k2*m.Z[t,'B']
    rc = 3*k2*m.Z[t,'B']
    cao = 4.0
    vo = 240
    T1 = 35000*(298-m.X[t,'T'])
    T2 = 4*240*30.0*(m.X[t,'T']-305.0)
    T3 = m.X[t,'V']*(6500.0*k1*m.Z[t,'A']-8000.0*k2*m.Z[t,'B'])
    Den = (30*m.Z[t,'A']+60*m.Z[t,'B']+20*m.Z[t,'C'])*m.X[t,'V']+3500.0
    exprs = dict()
    exprs['A'] = ra+(cao-m.Z[t,'A'])/m.X[t,'V']
    exprs['B'] = rb-m.Z[t,'B']*vo/m.X[t,'V']
    exprs['C'] = rc-m.Z[t,'C']*vo/m.X[t,'V']
    exprs['T'] = (T1+T2+T3)/Den
    exprs['V'] = vo
    return exprs
```

Where “m.X[t,'V']” and “m.X[t,'T']” are the additional state variables and “m.Z[t,component]” is the concentration of the component at time t. We can then simulate the model (or use experimental data if available and estimate the parameters) in the same way as described in the previous examples. Please follow the rest of the code and run the examples to obtain the output.

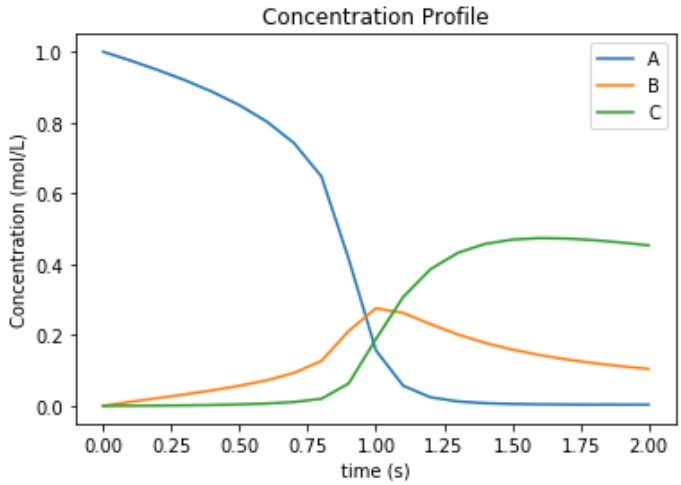


Figure 9: *Output of Tutorial example 3*

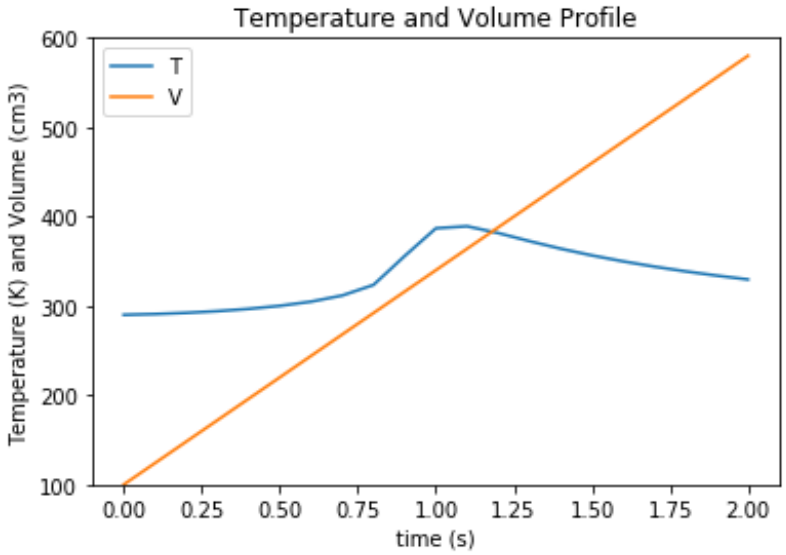


Figure 10: *Output of Tutorial example 3*

4.5 Tutorial 4 – Simulation of Advanced Reaction system with Algebraic equations

Now that complementary states are understood we can explain perhaps the most conceptually difficult part in KIPET, the idea of algebraic variables. The terms algebraics and algebraic variables are used in KIPET when referring to equations and variables in larger models that can be used to determine the ODEs where we have a number of states and equations. This can be illustrated with the Aspirin case study from Chen et al. (2016) where we have the more complex reaction mechanism:

$$\begin{aligned}
&\mathrm{SA} + \mathrm{AA} \xrightarrow{k_1} \mathrm{ASA} + \mathrm{HA} \\
&\mathrm{ASA} + \mathrm{AA} \xrightarrow{k_2} \mathrm{ASAA} + \mathrm{HA} \\
&\mathrm{ASAA} + \mathrm{H_2O} \xrightarrow{k_3} \mathrm{ASA} + \mathrm{HA} \\
&\mathrm{AA} + \mathrm{H_2O} \xrightarrow{k_4} 2\mathrm{HA} \\
&\mathrm{SA}(s) \xrightarrow{k_d} \mathrm{SA}(l) \\
&\mathrm{ASA}(l) \xrightarrow{k_c} \mathrm{ASA}(s)
\end{aligned} \tag{34}$$

With the rate laws being:

$$\begin{aligned}
r_1 &= k_1 c_{SA}(t) c_{AA}(t) \\
r_2 &= k_2 c_{ASA}(t) c_{AA}(t) \\
r_3 &= k_3 c_{ASAA}(t) c_{H_2O}(t) \\
r_4 &= k_4 c_{AA}(t) c_{H_2O}(t) \\
r_d &= \begin{cases} k_d \left(c_{SA}^{sat}(T) - c_{SA}(t) \right)^d, \text{if } m_{SA}(t) \geq 0 \\ 0, \text{if } m_{SA}(t) < 0 \end{cases} \\
r_g &= k_c \left(\max \left(c_{ASA}(t) - c_{ASA}^{sat}(T), 0 \right) \right)^c
\end{aligned} \tag{35}$$

And these can then be used to describe the concentrations of the liquid and solid components with the ODEs:

$$\dot{m}_{SA} = -M_{SA} V r_d, \dot{c}_{SA} = r_d - r_1 - \frac{\dot{V}}{V} c_{SA}$$

$$\dot{c}_{AA} = -r_1 - r_2 - r_4 - \frac{\dot{V}}{V} c_{AA}$$

$$\dot{c}_{HA} = r_1 + r_2 + r_3 + 2r_4 - \frac{\dot{V}}{V} c_{HA}$$

$$\dot{m}_{ASA} = M_{ASA} V r_g, \dot{c}_{ASA} = r_1 - r_2 + r_3 - r_g - \frac{\dot{V}}{V} c_{ASA} \qquad (36)$$

$$\dot{c}_{ASAA} = r_2 - r_3 - \frac{\dot{V}}{V} c_{ASAA}$$

$$\dot{c}_{H_2O} = -r_3 - r_4 + \frac{f}{V} c^{in}_{H_2O} - \frac{\dot{V}}{V} c_{H_2O}$$

$$\dot{V} = V \sum_{i=1}^{ns} v_i \left(\sum_{j=1}^{4} \gamma_{i,j} r_j + \gamma_{i,d} r_d + \gamma_{i,c} r_c + \varepsilon_i \frac{f}{V} c^{in}_{H_2O} \right)$$

This example can be described by the equations 35 (which are the “algebraics” in KIPET) and the ODEs, equations 36. which will then be the ODEs defining the system, making use of the reaction rate laws from the algebraics.

Translating these equations into code for KIPET we get the file found in Ex_4_sim_aspirin. In this example we need to declare new sets of states in addition to our components and parameters, as with Tutorial 3:

```
extra_states = dict()
extra_states['V'] = 0.0202
extra_states['Masa'] = 0.0
extra_states['Msa'] = 9.537

builder.add_complementary_state_variable(extra_states)
```

with the initial values given. In addition we can declare our algebraic variables (the rate variables and other algebraics):

```
algebraics = ['f','r0','r1','r2','r3','r4','r5','v_sum','Csat']
builder.add_algebraic_variable(algebraics)
```

Where f represents the addition of liquid to the reactor during the batch reaction.

For the final equation in the model (Equn 36) we also need to define the stoichiometric coefficients, gammas, and the epsilon for how the added water affects the changes in volume.

```
gammas = dict()
gammas['SA']=    [-1, 0, 0, 0, 1, 0]
```

```
gammas['AA']=    [-1,-1, 0,-1, 0, 0]
gammas['ASA']=   [ 1,-1, 1, 0, 0,-1]
gammas['HA']=    [ 1, 1, 1, 2, 0, 0]
gammas['ASAA']=  [ 0, 1,-1, 0, 0, 0]
gammas['H2O']=   [ 0, 0,-1,-1, 0, 0]

epsilon = dict()
epsilon['SA']= 0.0
epsilon['AA']= 0.0
epsilon['ASA']= 0.0
epsilon['HA']= 0.0
epsilon['ASAA']= 0.0
epsilon['H2O']= 1.0
partial_vol = dict()
partial_vol['SA']=0.0952552311614
partial_vol['AA']=0.101672206869
partial_vol['ASA']=0.132335206093
partial_vol['HA']=0.060320218688
partial_vol['ASAA']=0.186550717015
partial_vol['H2O']=0.0883603912169
```

To define the algebraic equations in Equn (35) we then use:

```
def rule_algebraics(m,t):
    r = list()
    r.append(m.Y[t,'r0']-m.P['k0']*m.Z[t,'SA']*m.Z[t,'AA'])
    r.append(m.Y[t,'r1']-m.P['k1']*m.Z[t,'ASA']*m.Z[t,'AA'])
    r.append(m.Y[t,'r2']-m.P['k2']*m.Z[t,'ASAA']*m.Z[t,'H2O'])
    r.append(m.Y[t,'r3']-m.P['k3']*m.Z[t,'AA']*m.Z[t,'H2O'])
    # dissolution rate
    step = 1.0/(1.0+exp(-m.X[t,'Msa']/1e-4))
    rd = m.P['kd']*(m.P['Csa']-m.Z[t,'SA']+1e-6)**1.90*step
    r.append(m.Y[t,'r4']-rd)
    #r.append(m.Y[t,'r4'])
    # crystalization rate
    diff = m.Z[t,'ASA'] - m.Y[t,'Csat']
    rc = 0.3950206559*m.P['kc']*(diff+((diff)**2+1e-6)**0.5)**1.34
    r.append(m.Y[t,'r5']-rc)
    Cin = 39.1
    v_sum = 0.0
    V = m.X[t,'V']
    f = m.Y[t,'f']
    for c in m.mixture_components:
           v_sum += partial_vol[c]*(sum(gammas[c][j]*m.Y[t,'r{}'.format(j)]
                   for j in range(6))+ epsilon[c]*f/V*Cin)
    r.append(m.Y[t,'v_sum']-v_sum)

    return r
builder.set_algebraics_rule(rule_algebraics)
```

Where the algebraics are given the variable name m.Y[t,'r1']. We can then use these algebraic equations to define our system of ODEs:

```
    def rule_odes(m,t):
        exprs = dict()

        V = m.X[t,'V']
        f = m.Y[t,'f']
        Cin = 41.4
        # volume balance
        vol_sum = 0.0
        for c in m.mixture_components:
             vol_sum += partial_vol[c]*(sum(gammas[c][j]*m.Y[t,'r{}'.format(j)]
                        for j in range(6))+ epsilon[c]*f/V*Cin)
        exprs['V'] = V*m.Y[t,'v_sum']

        # mass balances
        for c in m.mixture_components:
            exprs[c] = sum(gammas[c][j]*m.Y[t,'r{}'.format(j)] for j in
                        range(6))+ epsilon[c]*f/V*Cin - m.Y[t,'v_sum']*m.Z[t,c]

        exprs['Masa'] = 180.157*V*m.Y[t,'r5']
        exprs['Msa'] = -138.121*V*m.Y[t,'r4']
        return exprs

    builder.set_odes_rule(rule_odes)
    model = builder.create_pyomo_model(0.0,210.5257)
```

The rest can then be defined in the same way as other simulation problems. Note that in this problem the method for providing initializations from an external file is also shown with the lines:

```
 dataDirectory =
os.path.abspath(os.path.dirname( os.path.abspath( inspect.getfile(inspect.curre
ntframe() ) ) ))
    filename_initZ = os.path.join(dataDirectory, 'init_Z.csv')#Use absolute
      paths
    initialization = pd.read_csv(filename_initZ,index_col=0)
    sim.initialize_from_trajectory('Z',initialization)
   filename_initX = os.path.join(dataDirectory, 'init_X.csv')#Use absolute
      paths

    initialization = pd.read_csv(filename_initX,index_col=0)
    sim.initialize_from_trajectory('X',initialization)
    filename_initY = os.path.join(dataDirectory, 'init_Y.csv')#Use absolute
      paths
    initialization = pd.read_csv(filename_initY,index_col=0)
    sim.initialize_from_trajectory('Y',initialization)
```

where the external files are the csv's and the option index_col is from pandas and refers to the column to use for the labels. Following this, external files are also used for the flow of water fed into the reactor, as well as the saturation concentrations of SA and ASA (functions of temperature, calculated externally).

```
    dataDirectory = os.path.abspath(
        os.path.join( os.path.dirname( os.path.abspath( inspect.getfile(
            inspect.currentframe() ) ) ), 'data_sets'))
    traj =  os.path.join(dataDirectory,'extra_states.txt')

    dataDirectory = os.path.abspath(
        os.path.join( os.path.dirname( os.path.abspath( inspect.getfile(
            inspect.currentframe() ) ) ), 'data_sets'))
    conc =  os.path.join(dataDirectory,'concentrations.txt')

    fixed_traj = read_absorption_data_from_txt(traj)
    C = read_absorption_data_from_txt(conc)

    sim.fix_from_trajectory('Y','Csat',fixed_traj)
sim.fix_from_trajectory('Y','f',fixed_traj)
```

4.6 Tutorial 5 – Advanced reaction systems with additional states using finite element by finite element approach

Another functionality within KIPET is to use a finite element by element approach to initialize a problem. If you consider a fed-batch process, certain substances are added during the process in a specific manner dependent on time. This can be modeled using additional algebraic and state variables, similar to the process shown in Tutorial 4. In this tutorial, the following reaction system is simulated. Both the reaction system and the DAE system are shown below:

$$\frac{dV}{dt} = \begin{cases} const\ flowrate, t < 3.5h \\ 0, t > 3.5h \end{cases}$$

$$\frac{dc_{AH}}{dt} = -r_1 - r_3 - \frac{\dot{V}}{V} c_{AH}$$

$$\frac{dc_B}{dt} = -r_1 + r_4 - \frac{\dot{V}}{V} c_B$$

$$\frac{dc_{A^-}}{dt} = r_1 - r_2 + r_{-2} + r_3 - \frac{\dot{V}}{V} c_{A^-}$$

$$\frac{dc_{BH^+}}{dt} = r_1 - r_4 - \frac{\dot{V}}{V} c_{BH^+}$$

$$\frac{dc_C}{dt} = -r_2 + r_{-2} - \frac{\dot{V}}{V} c_C + \begin{cases} m_{C\_add} / V / 3.5, t < 3.5h \\ 0, t > 3.5h \end{cases}$$

$$\frac{dc_{AC^-}}{dt} = r_2 - r_{-2} - r_3 - r_4 - \frac{\dot{V}}{V} c_{AC^-}$$

$$\frac{dc_P}{dt} = r_3 + r_4 - \frac{\dot{V}}{V} c_P$$

$$\begin{aligned} AH + B &\xrightarrow{k_1} A^- + BH^+ & r_1 &= k_1 c_{AH} c_B \\ A^- + C &\xrightarrow{k_2} AC^- & r_2 &= k_2 c_{A^-} c_C \\ AC^- &\xrightarrow{k_{-2}} A^- + C & r_{-2} &= k_{-2} c_{AC^-} \\ AC^- + AH &\xrightarrow{k_3} P + A^- & r_3 &= k_3 c_{AC^-} c_{AH} \\ AC^- + BH^+ &\xrightarrow{k_4} P + B & r_4 &= k_4 c_{AC^-} c_{BH^+} \end{aligned}$$

The file for this tutorial is Ex_5_sim_fe_by_fe.py. For using the finite element by finite element approach you have to import the following package

```
from KIPET.library.FESimulator import *
```

In the case of having 5 rate laws, you will have 5 algebraic variables but an extra algebraic variable can be added which basically works as an input, such that you have 6 in total.

```
# add algebraics
algebraics = [0, 1, 2, 3, 4, 5]  # the indices of the rate rxns
builder.add_algebraic_variable(algebraics)
```

Then additional state variables can be added, which in this example is one additional state variable which models the volume.

```
# add additional state variables
extra_states = dict()
extra_states['V'] = 0.0629418
```

This is then included in the system of ordinary differential equations.

```
def rule_odes(m, t):
        exprs = dict()
        eta = 1e-2
         step = 0.5 * ((m.Y[t, 5] + 1) / ((m.Y[t, 5] + 1) ** 2 + eta ** 2) **
                 0.5 + (210.0 - m.Y[t,5]) / ((210.0 - m.Y[t, 5]) ** 2 + eta **
                       2) ** 0.5)
        exprs['V'] = 7.27609e-05 * step
        V = m.X[t, 'V']
        # mass balances
        for c in m.mixture_components:
            exprs[c] = sum(gammas[c][j] * m.Y[t, j] for j in m.algebraics if j !
                                    = 5) - exprs['V'] / V * m.Z[t, c]

            if c == 'C':
                  exprs[c] += 0.02247311828 / (m.X[t, 'V'] * 210) * step
        return exprs
```

Please be aware that the step equation and its application to the algebraic variable and equation m.Y[t,5] will act as a switch for the equations that require an action at a specific time.

In order to use the fe_factory to initialize the PyomoSimulator, we can use FESimulator, which automatically sets up the fe_factory problem using the data set up in KIPET's TemplateBuilder and then calls the PyomoSimulator to construct the simulation model. Similar to PyomoSimulator we first call FESimulator using:

```
sim = FESimulator(model)
```

And define the discretization scheme:

```
sim.apply_discretization('dae.collocation', nfe=50, ncp=3, scheme='LAGRANGE-
                                                              RADAU')
```

It is then necessary to declare the inputs_sub which shows which variable acts as the input. And also to fix the values of the input variables time measurement points for the simulation.

```
inputs_sub = {}
inputs_sub['Y'] = [5]
for key in sim.model.time.value:

    sim.model.Y[key, 5].set_value(key)
    sim.model.Y[key, 5].fix()
```

Finally we call the fe_factory using the FESimulator. When this function is called, it automatically runs the finite element by finite element march forward along the elements, as well as automatically patching the solutions to the PyomoSimulator model, thereby providing the initializations for it.

```
init = sim.call_fe_factory(inputs_sub)
```

Following this, we can call the PyomoSimulator function, run_sim(), as before in order to provide us with the final solution for the simulation, which should provide the ouputs, Figures 11 and 12.

An example showing how fe_factory can be called directly within KIPET is also given in the file Ad_7_sim_fe_by_fe_detailed.py. This approach should not be required, however provides useful insight into the mechanisms of fe_factory.

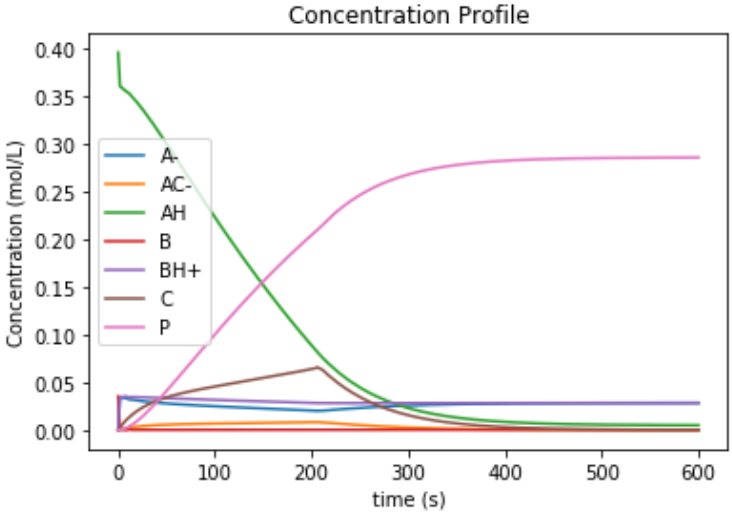


Figure 11: *Concentration profile of solution to Tutorial 5*

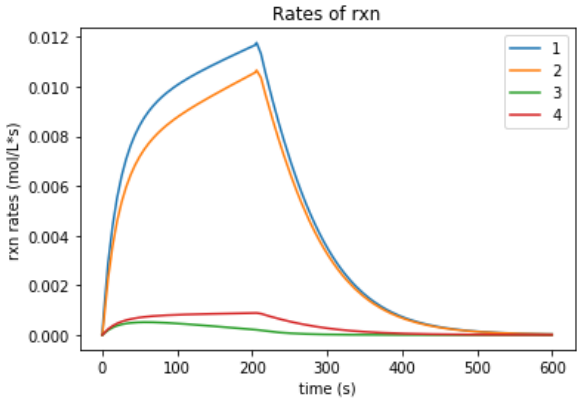


Figure 12: *Algebraic state profiles of solution to Tutorial 5*

4.7 Tutorial 6 – Reaction systems with known non-absorbing species in advance

If you are aware of which species are non-absorbing in your case in advance, you can exclude them from the identification process, fixing the associated column in the S-matrix to zero, and also excluding its variance.

You declare your components as in the examples above and then additionally declare the non-absorbing species by the following lines. If species ‘C’ is non-absorbing

```
non_abs = ['C']
builder.set_non_absorbing_species(opt_model, non_abs)
```

You can find an example for this in the examples folder called “Ex_6_non_absorbing.py”.

In the plot of the absorbance profile the non-absorbing species then remains zero as you can see in the following results.

```
Confidence intervals:
k2 (0.9999997318555397,1.0000000029408624)
k1 (0.09999999598268668,0.10000000502792096)

The estimated parameters are:
k2 0.999999867398201
k1 0.10000000050530382
```

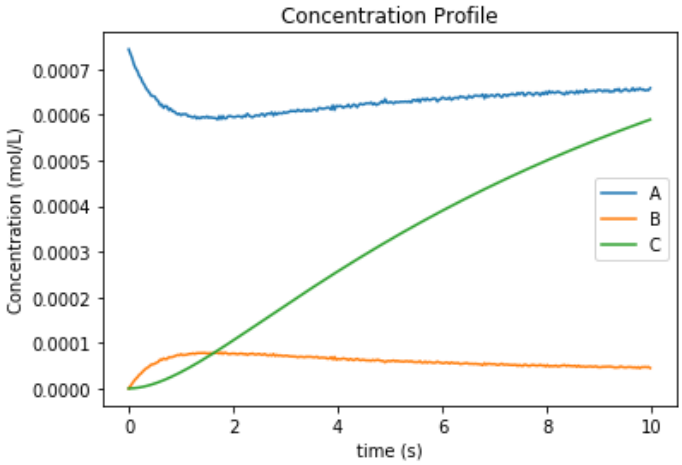


Figure 13: *Concentration profile of solution to Tutorial 6*

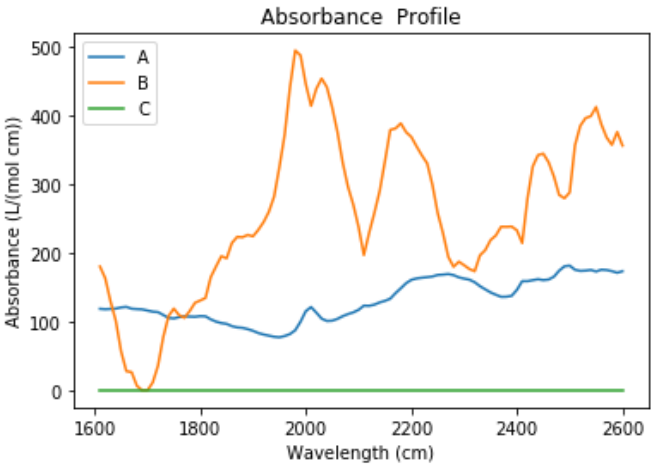


Figure 14: *Absorbance profile of Tutorial 6*

4.8 Tutorial 7– Parameter Estimation using concentration data

KIPET provides the option to also input concentration data in order to perform parameter estimation. The first term in the objective function (equation 17) is disabled in order to achieve this, so the problem essentially becomes a least squares minimization problem. The example, "Ex_7_concentration_input.py", shows how to use this feature. First, the data needs to be read and formatted using the data_tools function:

```
C_frame = read_concentration_data_from_txt(filename)
```

Following the TemplateBuilder call and parameter definition we can then add the concentration data to the model:

```
builder.add_concentration_data(C_frame)
```

After these two lines, the parameter estimation problem can be completed as shown above. If the user is interested in analyzing the confidence intervals associated with each estimated parameter, the same procedure as shown previously is used. An example of how this is called is found in "Ex_7_conc_input_conf.py".

That concludes the basic tutorials with the types of problems and how they can be solved. Provided in Table 2 is a list of the additional examples and how they differ. While this section not only acts as a tutorial, it also shows a host of the most commonly used functions in KIPET and how they work and which arguments they take. In the next section additional functions that are included in KIPET are explained, as well as any information regarding the functions discussed in the tutorials is also included.

4.9 Tutorial 8 – Time-dependent inputs of different kind using finite element by finite element approach

For modeling fed-batch processes, KIPET provides the option to add inputs due to dosing for certain species. For this the finite element by element approach is used to initialize the problem. If you consider a fed-batch process, certain substances are added during the process in a specific manner dependent on time. This can be modeled using additional algebraic and state variables, similar to the process shown in Tutorial 4.6. In this tutorial, the same reaction system as in Tutorial 4.6 is simulated. However, now we have discrete inputs for some substances.

An example of how this is realized with just one discrete input can be found in "Ex_5_sim_fe_by_fe_jump.py".

You have to add the time points where dosing takes place to a new set called feed_times:

```
feed_times=[100.]
builder.add_feed_times(feed_times)
```

Here we add a time point at 100.0 to the set feed_times which is then added to the model.

It is important that you add these additional time point(s) before you add the spectra or concentration data to the model.

Before you call fe_factory, you specify the components and the amount as well as the corresponding time points where dosing takes place in the following way:

```
Z_step = {'AH': .3} #Which component and which amount is added
jump_states = {'Z': Z_step}
jump_points = {'AH': 100.}#Which component is added at which point in time
jump_times = {'Z': jump_points}
```

Then you call fe_factory by providing those as additional arguments:

```
init = sim.call_fe_factory(inputs_sub, jump_states, jump_times, feed_times)
```

In case you want to use multiple inputs and also use dosing for algebraic components, you can find an example in “Ex_5_sim_fe_by_multjumpsandinputs.py”. A slightly modified version of the reaction system as in Tutorial 4.6 is implemented here. Here, the kinetic parameter k4 is also temperature dependent which is modeled by Arrhenius law.

The syntax for adding the specification of the components, the feeding amount and the time points, where dosing takes place, looks like this:

```
Z_step = {'AH': .3, 'A-': .1} #Which component and which amount is added
X_step = {'V': .01}
jump_states = {'Z': Z_step, 'X': X_step}
jump_points1 = {'AH': 101.035, 'A-': 400.}#Which component is added at which
                                          point in time
jump_points2 = {'V': 303.126}
jump_times = {'Z': jump_points1, 'X': jump_points2}
```

Example “Ex_5_sim_fe_by_multjumpsandinputs.py” also shows how to use discrete trajectories as inputs. In this case temperature values are read from a file where the fix_from_trajectory function interpolates in between the provided values as well.

The inputs can be read from a txt or csv file via

```
read_absorption_data_from_txt(Ttraj)
```

or

```
read_absorption_data_from_csv(Ttraj),
```

where Ttraj should load a file including the values in the right format as already explained earlier.

Then, an input for the algebraic state variable Temp can be fixed by calling the fix_from_trajectory function:

```
inputs_sub = {}
inputs_sub['Y'] = ['5', 'Temp']
sim.fix_from_trajectory('Y', 'Temp', fixed_Ttraj)
```

Since the model can not discriminate inputs from other algebraic elements, we still need to define the inputs as inputs_sub.

We have to do all this before we call fe_factory as above.

4.10 Tutorial 9 – Variance and parameter estimation with time-dependent inputs using finite element by finite element approach

In case of dealing with fed-batch processes as in Tutorial 8, KIPET provides the capabilities of also performing variance and parameter estimation for those kind of problems. For this the finite element by element model is used as the optimization model as well.

An example of how this is realized with spectral data can be found in "Ex_2_estimationfefactoryTempV.py". This example uses the reaction mechanism introduced in Tutorial 1 but now with temperature dependence of the parameter k2 modeled by the Arrhenius law. Furthermore, volume change takes place here. You first run the simulation as in Tutorial 8. For the optimization in addition you have to declare additional arguments, such as algebraic variables that are fixed from a trajectory or that are fixed to certain keys.

You declare your inputs by:

```
inputs_sub = {}
inputs_sub['Y'] = ['3','Temp'].
```

Then, you declare more optional arguments regarding these inputs:

```
trajs = dict()
trajs[('Y', 'Temp')] = fixed_Ttraj
fixedy = True  # instead of things above
fixedtraj = True
yfix={}
yfix['Y']=['3'] #needed in case of different input fixes
yfixtraj={}
yfixtraj['Y']=['Temp']
```

Thereby,

```
fixedy = True
```

should be set if you have inputs of this kind

```
for key in sim.model.time.value:
      sim.model.Y[key, '3'].set_value(key)
      sim.model.Y[key, '3'].fix()
```

in combination with setting

```
yfix={}
yfix['Y']=['3']  #needed in case of different input fixes.
```

In case of dealing with inputs that are fixed from trajectories, such as

```
trajs = dict()
trajs[('Y', 'Temp')] = fixed_Ttraj
fixedtraj = True
```

All of these arguments are handed later to the VarianceEstimator or ParameterEstimator via:

```
results_variances = v_estimator.run_opt('ipopt',
                                        tee=True,
                                        solver_options=options,
                                        tolerance=1e-5,
                                        max_iter=15,
                                        subset_lambdas=A_set,
                                        inputs_sub=inputs_sub,
                                        trajectories=trajs,
                                        jump=True,
                                        jump_times=jump_times,
                                        jump_states=jump_states,
                                        fixedy=True,
                                        fixedtraj=True,
                                        yfix=yfix,
                                        yfixtraj=yfixtraj,
                                        feed_times=feed_times
                                        )
```

or

```
results_pyomo = p_estimator.run_opt('k_aug',
                                    tee=True,
                                    solver_opts=options,
                                    variances=sigmas,
                                    with_d_vars=True,
                                    covariance=True,
                                    inputs_sub=inputs_sub,
                                    trajectories=trajs,
                                    jump=True,
                                    jump_times=jump_times,
                                    jump_states=jump_states,
                                    fixedy=True,
                                    fixedtraj=True,
                                    yfix=yfix,
                                    yfixtraj=yfixtraj,
                                    feed_times=feed_times)
```

As the parameter values are fixed when running the simulation first. You have to change this before running the Variance Estimation and the Parameter Estimation via:

```
model=builder.create_pyomo_model(0.0,10.0)
#Now introduce parameters as non fixed
model.del_component(params)
builder.add_parameter('k1',bounds=(0.0,5.0))
builder.add_parameter('k2Tr',0.2265)
builder.add_parameter('E',2.)
model = builder.create_pyomo_model(0, 10)
v_estimator = VarianceEstimator(model)
```

There are two important things that you should keep in mind. You have to add the feed points before adding the dataset of either concentration data or spectral data to the model.

Furthermore, you should always check the feed times and points carefully, such that they match and the right values for the arguments above are provided.

Everything else works as explained for the general estimation cases like for example "Ex_2_estimation_conf.py" and as explained for the example with inputs in Tutorial 8.

In addition to this, "Ad_5_conc_in_input_conf.py" provides an example for parameter estimation with concentration data instead of spectral data.

4.11 Tutorial 10 – Using k_aug to obtain confidence intervals

This can be done using the new package developed by David M. Thierry called k_aug, which computes the reduced hessian instead of sIpopt. In order to use this instead of sIpopt, when calling the solver, the solver needs to be set to be 'k_aug'. All other steps are the same as in previous examples. The examples that demonstrate this functionality are "Ex_7_conc_input_conf_k_aug.py" and "Ex_2_estimation_conf_k_aug.py".

```
results_pyomo = p_estimator.run_opt('k_aug',    tee=True,
                                    solver_opts = options,
                                    variances=sigmas,
                                    with_d_vars=True,
                                    covariance=True)
```

4.12 Tutorial 11 – Interfering species and fixing absorbances

If we know in advance that one of the absorbing species does not react in advance, we are able to easily include this by merely adding the component to the model as with all other species and including the ODE as follows (see example "Ex_2_abs_not_react"):

```
exprs['D'] = 0
```

In this example we obtain the following profile and absorbances:

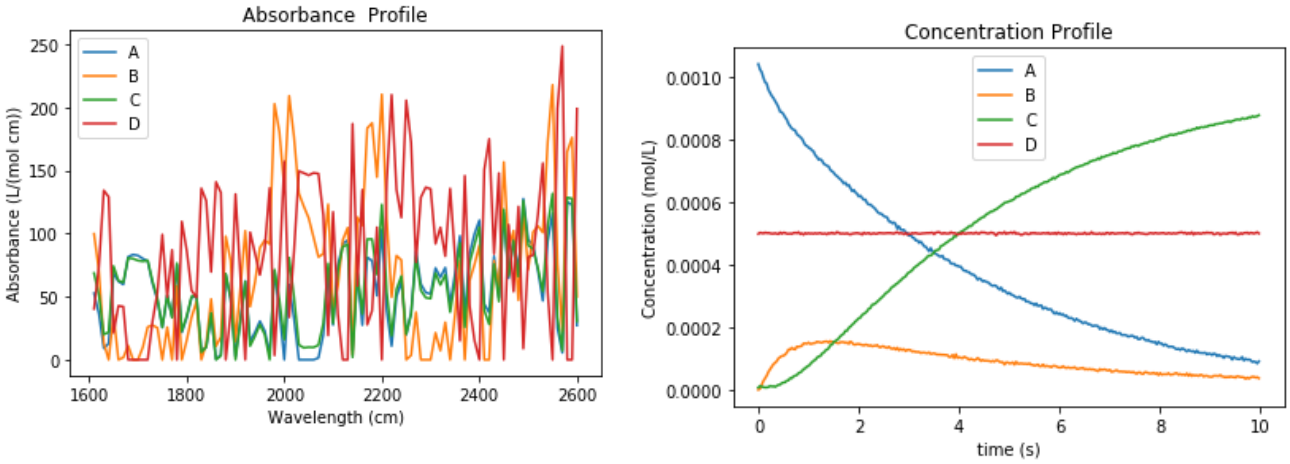


Figure 15: *Absorbance profiles and concentration profiles of Tutorial problem 11a.*

If the user knows, in advance, the absorbance profile of a specific component then we can also fix this absorbance. This is shown in “Ex_2_abs_known_non_react.py” where we use the following function.

```
known_abs = ['D']
builder.set_known_absorbing_species(opt_model, known_abs, S_frame)
```

where S_frame is a pandas dataframe containing the the species’ absorbance profile and opt_model is the pyomo model as shown in the example. From this we are able to run the VarianceEstimator and ParameterEstimator to obtain the solutions shown in Figure 16.

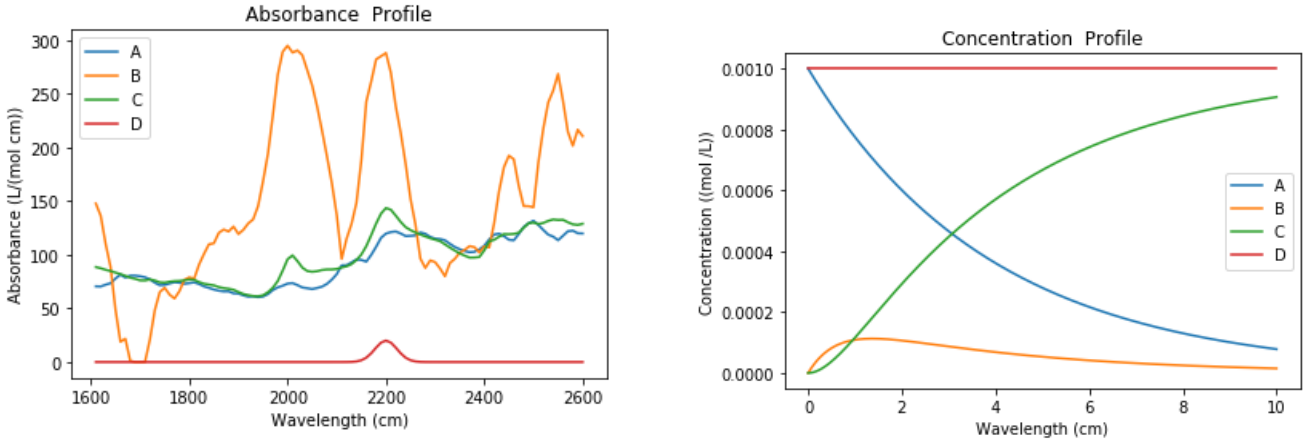


Figure 16: *Absorbance profiles and concentration profiles of Tutorial problem 11b.*

Care should be taken when fixing species’ absorbance profiles, however as this reduces the degrees of freedom for the problem, resulting in issues when obtaining the confidence intervals, in particular.

4.13 Tutorial 12 – Estimability analysis

The EstimabilityAnalyzer module is used for all algorithms and tools pertaining to estimability. Thus far, estimability analysis tools are only provided for cases where concentration data is available. The methods rely on k_aug to obtain sensitivities, so will only work if k_aug is installed and added to path. The example from the example directory is “Ex_8_estimability.py”.

After setting up the model in TemplateBuilder, we can now create the new class:

```
e_analyzer = EstimabilityAnalyzer(opt_model)
```

It is very important to apply discretization before running the parameter ranking function.

```
e_analyzer.apply_discretization('dae.collocation',nfe=60,ncp=1,
scheme='LAGRANGE-RADAU')
```

The algorithm for parameter ranking requires the definition by the user of the confidences in the parameter initial guesses, as well as measurement device error in order to scale the sensitivities obtained. In order to run the full optimization problem, the variances for the model are also still required, as in previous examples.

```
param_uncertainties = {'k1':0.09,'k2':0.01,'k3':0.02,'k4':0.01}
sigmas = {'A':1e-10,'B':1e-10,'C':1e-11, 'D':1e-11,'E':1e-11,'device':3e-9}
meas_uncertainty = 0.01
```

The parameter ranking algorithm from Yao, et al. (2003) needs to be applied first in order to supply a list of parameters that are ranked. This algorithm ranks parameters using a sensitivity matrix computed from the model at the initial parameter values (in the middle of the bounds automatically, or at the initial guess provided the user explicitly). This function is only applicable to the case where you are providing concentration data, and returns a list of parameters ranked from most estimable to least estimable. Once these scalings are defined we can call the ranking function:

```
listparams = e_analyzer.rank_params_yao(meas_scaling = meas_uncertainty,
                        param_scaling = param_uncertainties, sigmas =sigmas)
```

This function returns the parameters in order from most estimable to least estimable. Finally we can use these ranked parameters to perform the estimability analysis methodology suggested by Wu, et al. (2011) which uses an algorithm where a set of simplified models are compared to the full model and the model which provides the smallest mean squared error is chosen as the optimal number of parameters to estimate. This is done using:

```
params_to_select = e_analyzer.run_analyzer(method = 'Wu', parameter_rankings =
            listparams, meas_scaling = meas_uncertainty, variances = sigmas)
```

This will return a list with only the estimable parameters returned. All remaining parameters (non-estimable) should be fixed at their most likely values.

For a larger example with more parameters and which includes the data generation, noising of data, as well as the application of the estimability to a final parameter estimation problem see “Ex_9_estimability_with_problem_gen.py”

```
sigmas = {'A':1e-10,'B':1e-10,'C':1e-11, 'D':1e-11,'E':1e-11,'device':3e-9}
meas_uncertainty = 0.01
```

The parameter ranking algorithm from Yao, et al. (2003) needs to be applied first in order to supply a list of parameters that are ranked. This algorithm ranks parameters using a sensitivity matrix computed from the model at the initial parameter values (in the middle of the bounds automatically, or at the initial guess provided the user explicitly). This function is only applicable to the case where you are providing concentration data, and returns a list of parameters ranked from most estimable to least estimable. Once these scalings are defined we can call the ranking function:

```
listparams = e_analyzer.rank_params_yao(meas_scaling = meas_uncertainty,
                  param_scaling = param_uncertainties, sigmas =sigmas)
```

This function returns the parameters in order from most estimable to least estimable. Finally we can use these ranked parameters to perform the estimability analysis methodology suggested by Wu, et al. (2011) which uses an algorithm where a set of simplified models are compared to the full model and the model which provides the smallest mean squared error is chosen as the optimal number of parameters to estimate. This is done using:

```
params_to_select = e_analyzer.run_analyzer(method = 'Wu', parameter_rankings =
            listparams, meas_scaling = meas_uncertainty, variances = sigmas)
```

This will return a list with only the estimable parameters returned. All remaining parameters (non-estimable) should be fixed at their most likely values.

For a larger example with more parameters and which includes the data generation, noising of data, as well as the application of the estimability to a final parameter estimation problem see "Ex_9_estimability_with_problem_gen.py"

4.14 Tutorial 13 – Using the wavelength selection tools

In this example we are assuming that we have certain wavelengths that do not contribute much to the model, rather increasing the noise and decreasing the goodness of the fit of the model to the data. We can set up the problem in the same way as in Example 2 and solve the full variance and parameter estimation problem with all wavelengths selected.

Note that in order to use the wavelength selection functions, it is important to make a copy of the TemplateBuilder prior to adding the spectral data. This is shown on lines 70 – 76. Here, we make a copy of the TemplateBuilder class after adding the model equations, but before the spectral data:

```
builder_before_data = builder
builder.add_spectral_data(D_frame)
end_time = 10
opt_model = builder.create_pyomo_model(0.0,end_time)
```

After completing the normal parameter estimation, we can determine the lack of fit with the following function

```
lof = p_estimator.lack_of_fit()
```

This returns the lack of fit as a percentage, in this case 1.37 % lack of fit. We can now determine which wavelengths have the most significant correlations to the concentration matrix predicted by the model:

```
correlations = p_estimator.wavelength_correlation()
```

This function prints a figure that shows the correlations (0,1) of each wavelngth in the output to the concentration profiles. As we can see from figure, some wavelengths are highly correlated, while others have little correlation to the model concentrations. Note that the returned *correlations* variable contains a dictionary (unsorted) with the wavelengths and their correlations. In order to print the figure, these need to be sorted and decoupled with the following code:

```
if with_plots:
    lists1 = sorted(correlations.items())
    x1, y1 = zip(*lists1)
    plt.plot(x1,y1)
    plt.xlabel("Wavelength (cm)")
    plt.ylabel("Correlation between species and wavelength")
    plt.title("Correlation of species and wavelength")
```

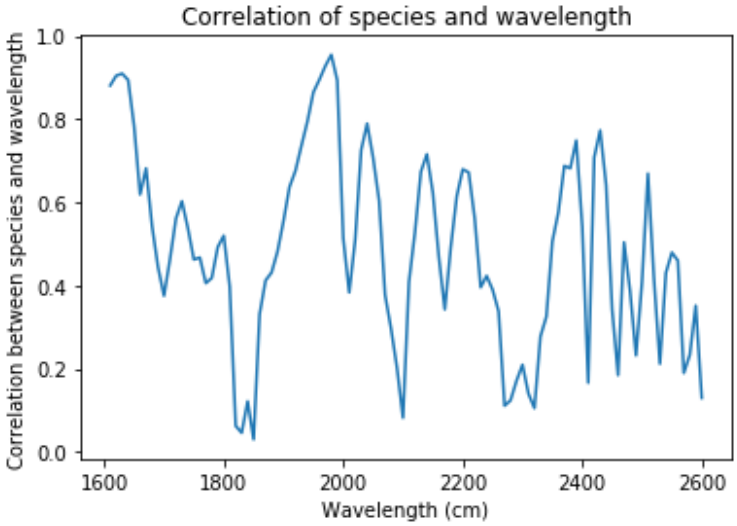


Figure 17: Wavelength correlations for the tutorial example 14.

We now have the option of whether to select a certain amount of correlation to cut off, or whether to do a quick analysis of the full correlation space, in the hopes that certain filter strengths will improve our lack of fit. Ultimately, we wish to find a subset of wavelengths that will provide us with the lowest lack of fit. In this example, we first run a lack of fit analysis that will solve, in succession, the parameter estimation problem with wavelengths of less than 0.2, 0.4, 0.6, and 0.8 correlation removed using the following function:

```
p_estimator.run_lof_analysis(builder_before_data, end_time, correlations,
                             lof, nfe, ncp, sigmas)
```

Where the arguments are builder_before_data (the copied TemplateBuilder before the spectral data is added), the end_time (the end time of the experiment), correlations (the dictionary of wavelngths and their correlations obtained above), lof (the lack of fit from the full parameter estimation problem, i.e. where all the wavelengths are selected), followed by the nfe (number of finite elements), ncp (number of collocation points), and the sigmas (variances from VarianceEstimator).

These are the required arguments for the function. The outputs are as follows:

```
When wavelengths of less than  0 correlation are removed
The lack of fit is:  1.3759210191412483
When wavelengths of less than  0.2 correlation are removed
The lack of fit is:  1.3902630158740596
When wavelengths of less than  0.4 correlation are removed
The lack of fit is:  1.4369628529062384
When wavelengths of less than  0.6000000000000001 correlation are removed
The lack of fit is:  1.4585991614309648
When wavelengths of less than  0.8 correlation are removed
The lack of fit is:  1.5927062320924816
```

From this analysis, we can observe that by removing many wavelengths we do not obtain a much better lack of fit, however, let us say that we would like to do a finer search between 0 and 0.12 filter on the correlations with a search step size of 0.01. We can do that with the following extra arguments:

```
p_estimator.run_lof_analysis(builder_before_data, end_time, correlations,
                             lof, nfe, ncp, sigmas, step_size = 0.01,
                             search_range = (0, 0.12))
```

With the additional arguments above, the output is:

```
When wavelengths of less than  0 correlation are removed
The lack of fit is:  1.3759210191412483
When wavelengths of less than  0.01 correlation are removed
The lack of fit is:  1.3759210099692445
When wavelengths of less than  0.02 correlation are removed
The lack of fit is:  1.3759210099692445
When wavelengths of less than  0.03 correlation are removed
The lack of fit is:  1.3759210099692445
When wavelengths of less than  0.04 correlation are removed
The lack of fit is:  1.3733116835623844
When wavelengths of less than  0.05 correlation are removed
The lack of fit is:  1.3701575988048247
When wavelengths of less than  0.06 correlation are removed
The lack of fit is:  1.3701575988048247
When wavelengths of less than  0.07 correlation are removed
The lack of fit is:  1.3681439750540936
When wavelengths of less than  0.08 correlation are removed
The lack of fit is:  1.3681439750540936
When wavelengths of less than  0.09 correlation are removed
The lack of fit is:  1.366438881909768
When wavelengths of less than  0.10 correlation are removed
The lack of fit is:  1.366438881909768
When wavelengths of less than  0.11 correlation are removed
The lack of fit is:  1.3678616037309008
When wavelengths of less than  0.12 correlation are removed
The lack of fit is:  1.370173019880385
```

So from this output, we can see that the best lack of fit is possibly somewhere around 0.095, so we could either refine our search or we could just run a single parameter estimation problem based on this specific wavelength correlation. In order to do this, we can obtain the data matrix for the parameter estimation by running the following function:

```
new_subs = wavelength_subset_selection(correlations = correlations, n =
                                                    0.095)
```

Which will just return the dictionary with all the correlations below the threshold removed. Finally, we run the ParameterEstimator on this new data set, followed by a lack of fit analysis, using:

```
results_pyomo = p_estimator.run_param_est_with_subset_lambdas
      (builder_before_data, end_time, new_subs, nfe, ncp, sigmas)
```

In this function, the arguments are all explained above and the outputs are the follows:

```
The lack of fit is  1.366438881909768  %
k2 0.9999999977885373
k1 0.22728234196932856
```

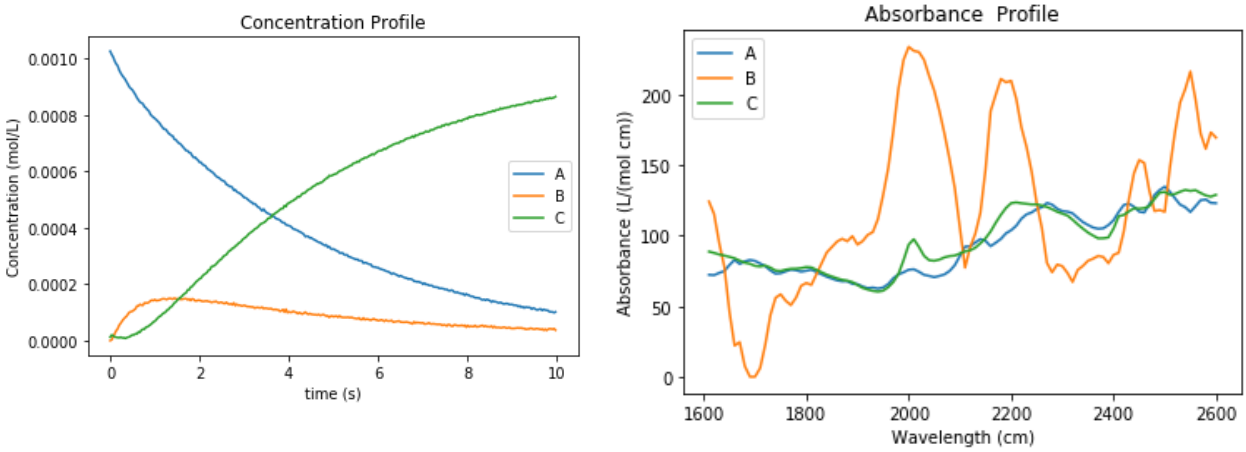


Figure 18: *Solution profiles for tutorial example 14.*

4.15 Tutorial 14 – Parameter estimation over multiple datasets

KIPET now also allows for the estimation of kinetic parameters with multiple experimental datasets through the MultipleExperimentsEstimator class. Internally, this procedure is performed by running the VarianceEstimator (optionally) over each dataset, followed by ParameterEstimator on individual models. After the local parameter estimation has been performed, the code blocks are used to initialize the full parameter estimation problem. The algorithm automatically detects whether parameters are shared across experiments based on their names within each model. Note that this procedure can be fairly time-consuming. In addition, it may be necessary to spend considerable time tuning the solver parameters in these problems, as the system involves the solution of large, dense linear systems in a block structure linked via equality constraints (parameters). It is advised to try different linear solver combinations with various IPOPT solver options if difficulty is found solving these. The problems may also require large amounts of RAM, depending on the size.

The first example we will look at in this tutorial is entitled “Ex_11_mult_exp_conc.py”, wherein we have a dataset that contains concentration data for a simple reaction and another dataset that is the same one with added noise using the following function:

```
C_frame2 = add_noise_to_signal(C_frame1, 0.0001)
```

We then define our model as we have done before. In contrast to previously, however, we create dictionaries containing the datasets, start and end times for the experiments, as well as the variances:

```
datasets = {'Exp1': C_frame1, 'Exp2': C_frame2}
start_time = {'Exp1':0.0, 'Exp2':0.0}
end_time = {'Exp1':10.0, 'Exp2':10.0}
sigmas = {'A':1e-10,'B':1e-10,'C':1e-10}
variances = {'Exp1':sigmas, 'Exp2':sigmas}
```

Notice here that we need to be consistent with labelling each dataset, as these are used internally to define the individual block names. Now we are ready to call the MultipleExperimentsEstimator class. When we do this, we define our class using the datasets as the argument. This ensures that we know the type of data and names for our separate blocks of data.

```
pest = MultipleExperimentsEstimator(datasets)
```

Instead of applying the discretization directly to the model, we can now just add it to the function as an argument. Notice here that this function performs all the steps of estimation in one go.

```
results_pest = pest.run_parameter_estimation(solver = 'ipopt',
                                             tee=True,
                                              nfe=nfe,
                                              ncp=ncp,
                                              solver_opts = options,
                                              start_time=start_time,
                                              end_time=end_time,
                                            spectra_problem = False,
                                              sigma_sq=variances,
                                                 builder = builder)
```

In the above code block, the builder can be either a dictionary of different models (labelled with the dataset labels) or a single model that applies to all datasets. The ‘spectra_problem’ argument is automatically set to True, so when concentration is provided, we need to set this to False.

After running this, we will obtain the results from both datasets separately and then a combined datasets solution at the end. Note that when printing solutions we now need to use the following notation to get the solutions from both blocks:

```
for k,v in results_pest.items():
    print(results_pest[k].P)
```

```
    if with_plots:
        for k,v in results_pest.items():
            results_pest[k].C.plot.line(legend=True)
            plt.xlabel("time (s)")
            plt.ylabel("Concentration (mol/L)")
            plt.title("Concentration Profile")

            results_pest[k].Z.plot.line(legend=True)
            plt.xlabel("time (s)")
            plt.ylabel("Concentration (mol/L)")
            plt.title("Concentration Profile")

            plt.show()
```

Where the dictionary results_pest contains each of the experimental datasets. The results are as follows:

```
The estimated parameters are:
k1    0.224999
k2    0.970227
```

Another example here is the example where we have multiple experimental datasets from spectroscopic data and we also perform variance estimation. This example is "Ex_11_estimation_mult_exp.py". In this example we have three datasets, each perturbed by different levels of noise for illustrative purposes. One dataset is also of different size, just to demonstrate KIPET's ability to do so:

```
    D_frame1 = decrease_wavelengths(D_frame1,A_set = 2)
```

This function removes every second wavelengths from the data matrix D_frame1. Similar to before, we add the three datasets separately into the MultipleExperiments class as a dictionary and finally run the variance estimation:

```
    results_variances = pest.run_variance_estimation(solver = 'ipopt',
                                                     tee=False,
                                                     nfe=nfe,
                                                     ncp=ncp,
                                                     solver_opts = options,
                                                     start_time=start_time,
                                                     end_time=end_time,
                                                     builder = builder)
```

This function runs the variance estimation on each block separately and automatically inserts them into the parameter estimation section, so there is no need to supply them if this is run prior to the parameter estimation. Finally we can run the parameter estimation as before:

```
    results_pest = pest.run_parameter_estimation(solver = 'ipopt',
                                                    tee=True,
                                                     nfe=nfe,
```

```
ncp=ncp,
solver_opts = options,
start_time=start_time,
end_time=end_time,
builder = builder)
```

This outputs the following:

The estimated parameters are:

k2 1.357178

k1 0.279039

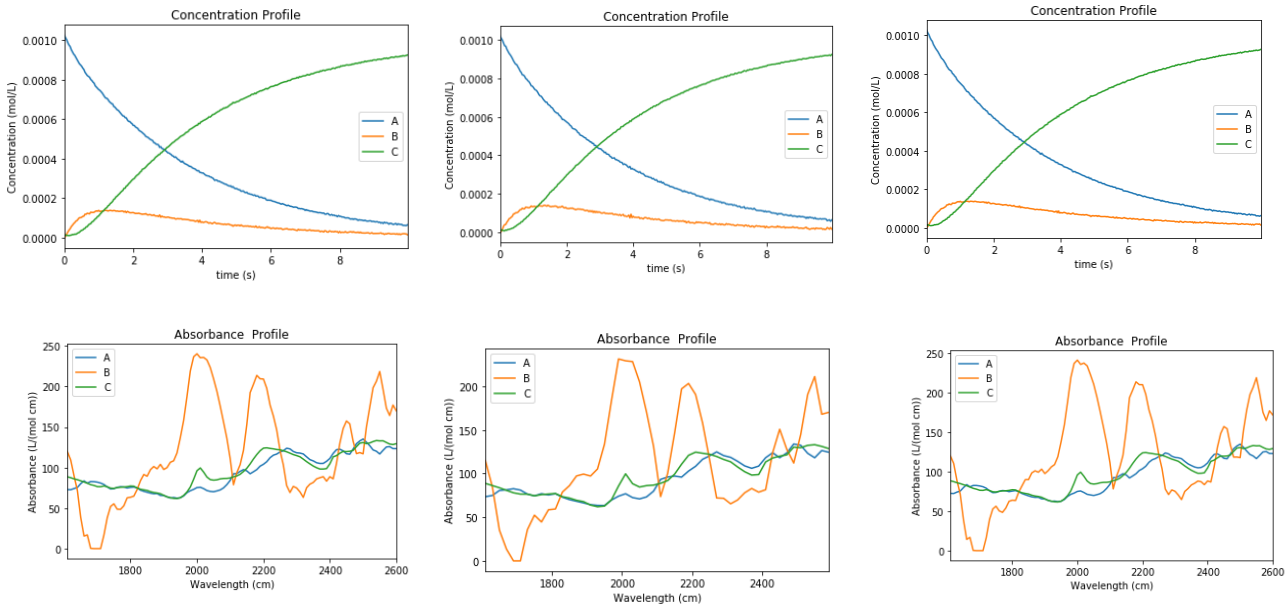


Figure 19: *Solution profiles for tutorial example 15.*

There are a number of other examples showing how to implement the multiple experiments across different models with shared global and local parameters as well as how to obtain confidence intervals for the problems.

It should be noted that obtaining confidence intervals can only be done when declaring a global model, as opposed to different models in each block. This is due to the construction of the covariance matrices. When obtaining confidence intervals for multiple experimental datasets it is very important to ensure that the solution obtained does not include irrationally large absorbances (from species with low or no concentration) and that the solution of the parameters is not at very close to a bound. This will cause the sensitivity calculations to be aborted, or may result in incorrect confidence intervals.

All the additional problems demonstrating various ways to obtain kinetic parameters from different experimental set-ups are shown in the example table and included in the folder with tutorial examples.

This concludes the last of the tutorial examples. This hopefully provides a good overview of the capabilities of the package and we look forward to getting feedback once these have been applied to your own problems. Table 2 on the following page provides a complete list of all of the example problems in the KIPET package, with some additional explanations.

The next section of the documentation provides more detailed and miscellaneous functions from within KIPET that were not demonstrated in the tutorials.

TABLE 2: *List of example problems*

Filename	**Example problem description**
Ex_1_ode_sim.py	Tutorial example of simulation (of reaction system 1, RS1)
Ex_2_estimation.py	Tutorial example of parameter estimation with variance estimation (of RS1)
Ex_2_estimation_conf.py	Tutorial example of parameter estimation problem above with variance estimation and confidence intervals from sIpopt (RS1)
Ex_2_estimation_conf_k_aug.py	Tutorial example of parameter estimation problem above with variance estimation and confidence intervals from k_aug (RS1)
Ex_2_estimation_filter_msc.py	Same problem as above with MSC followed by SG pre-processing
Ex_2_estimation_filter_snv.py	Same problem as above with SNV followed by SG pre-processing
Ex_2_estimationfefactoryTempV.py	Tutorial estimation for variance and parameter estimation with inputs (modified RS1)
Ex_2_abs_not_react.py	Tutorial example of parameter estimation where one species is absorbing but not reacting
Ex_2_abs_known_non_react.py	Tutorial example of parameter estimation where one species is absorbing and not reacting, however we know this species absorbance profile.
Ex_2_with_SVD	Example demonstrating how to use the basic_pca function
Ex_2_estimation_bound_prof_fixed_variance.py	Example demonstrating how to fix device variance and also how to bound and fix variable profiles.

Filename	Example problem description
Ex_3_complementary.py	Tutorial simulation that includes additional complementary states (RS2)
Ex_4_sim_aspirin.py	Tutorial simulation of an aspirin batch reactor (RS3) that shows how additional states and algebraics are used
Ex_5_sim_fe_by_fe_jump.py	Tutorial simulation of a large reaction system (RS4) including demonstration of the finite element by finite element initialization method. The reaction system is that of the Michael's reaction but here dosing takes place. That means for one of the species feeding takes place at one time point during the process. This example shows how dosing inputs can be realized (Section 4.9).
Ex_5_sim_fe_by_fe_multjumpsandinputs.py	Tutorial simulation of a large reaction system (RS4) including demonstration of the finite element by finite element initialization method. The reaction system is a slightly modified version of the one of the Michael's reaction. Here dosing takes place for multiple species that means for multiple species feeding takes place at different points in time during the process. Furthermore, this example shows how inputs via discrete trajectories can be realized. For this, one of the kinetic parameters is now assumed to be temperature dependent and temperature inputs are provided via temperature values read from a file (Section 4.9).
Ex_6_non_absorbing	Example of a problem where non-absorbing components are included.
Ex_7_concentration_input.py	Tutorial problem describing RS1 where concentration data is provided by the user.
Ex_7_conc_input_conf.py	Tutorial problem describing RS1 where concentration data is provided by the user and confidence intervals are shown.
Ex_8_estimability.py	Tutorial problem demonstrating the estimability analysis
Ex_9_estimability_with_prob_gen.py	Tutorial problem where problem generation is done via simulation, random normal noise is added, followed by estimability analysis and finally parameter estimation on reduced model.
Ex_10_estimation_lof_correlation.py	Tutorial problem 14 where subset selection is made based on the correlation between wavelengths and the species concentrations. We also introduce the lack of fit as a way to judge the selection.
Ex_11_estimation_mult_exp.py	Tutorial problem 15 with 2 spectroscopic datasets and shared parameters
Ex_11_estimation_mult_exp_conf.py	Tutorial problem 15 with 2 spectroscopic datasets and shared parameters with confidence intervals

Filename	Example problem description
Ex_12_estimation_mult_exp_conc.py	Tutorial 15 : parameter estimation with 2 concentration datasets.
Ex_12_estimation_mult_exp_conc_conf.py	Tutorial 15 : parameter estimation with 2 concentration datasets including confidence intervals
Ex_12_multexp_conc_diffreact.py	Parameter estimation with 2 concentration datasets, including data simulation with different initial conditions and including confidence intervals
Ad_1_estimation.py	Additional parameter estimation problem with known variances, but with a least squares optimization run to get initialization for the parameter estimation problem (Section 5.6). (RS1)
Ad_2_estimation_warmstart.py	Tutorial example of parameter estimation with variance estimation (of RS1) with warmstart option and estimating parameters in steps
Ad_2_ode_sim.py	Simulation of RS3 from Sawall, et al. (2012), nonlinear system
Ad_2_scaled_estimation.py	RS3 parameter estimator, including least squares initialization, variance estimator, and parameter estimation
Ad_3_sdae_sim_non_abs.py	RS1 system with generated absorbance data and a non-absorbing component, this problem generates absorbance data and then runs a simulation that generates a D-matrix for the parameter estimation and variance estimation
Ad_4_sdae_sim.py	Tutorial problem with inputted absorbances (RS1)
Ad_5_complementary_sim.py	RS3 with temperature included in simulation
Ad_5_conc_in_input_conf.py	Parameter estimation with inputs for RS3 with concentration data
Ad_6_sawall.py	Parameter estimation of another nonlinear reaction system from the Sawall, et al., 2012, paper.
Ad_7_sim_fe_by_fe_detailed.py	Example of using fe_factory explicitly within KIPET. (RS4)
Ad_8_conc_input_est_conf.py	RS1 with concentration data as the input. Parameters estimated with confidence intervals
Ad_9_conc_in_sawall_est.py	Parameter estimation on nonlinear reaction system from the Sawall, et al., 2012, paper with concentration data inputted.
Ad_10_aspirin_FESimulator.py	The aspirin example simulated using FESimulator

Filename	Example problem description
Ad_11_estimation_mult_exp_conf.py	Example with multiple datasets including reactions that do not occur in some datasets. Includes confidence intervals.

5. Additional Functionalities

Along with the functions explained in the examples, KIPET also provides users with a host of other functions that can be used. In this section some of the additional functions provided in KIPET are shown and detailed, along with some the other subtleties that may be useful to the user in implementing and improving their models.

5.1 Data manipulation tools

KIPET provides a variety of data manipulation tools and the way in which data is inputted is extremely important in order to correctly pass the results of the user's experiments onto KIPET. This section provides a further clarification on the types of files that KIPET is able to utilize, how the data should be arranged, as well as how to use KIPET to generate data.

5.1.1 Input matrices

Loading data (D) matrices

When using KIPET for parameter estimation it is important that the data matrix, D, be inputted correctly. KIPET can read both text files (.txt) or Comma Separated Value files (.csv) from software such as Microsoft Excel or OpenOffice. Examples of data sets and how best to format them are included in the folder "Examples/data_sets". Text files are best formulated with unlabeled columns with column 1 being time, column 2 being the wavelength, and column 3 being the absorbance associated with the specific wavelength. The method read_spectral_data_from_txt(filename.txt) automatically sorts the data and compiles it into a Pandas DataFrame for further manipulation by KIPET. Even though the order of the data is not important (except that rows need to be consistent), it is important to input the data as floating values and to separate the columns with a space.

When inputting a CSV file, it is necessary to label the columns with headings of the wavelength values and with row labels for the measuring times. The matrix will then be in the correct form with the absorption values being the entries. The read_spectral_data_from_csv(filename.csv) function is the function to call in this case. Additionally, if you have data directly outputted from an instrument as a CSV, there are some tools in this function that automatically turn timestamped data into seconds and also manipulate the matrix into KIPET's preferred format. This is done through the use of an additional argument "instrument = True". If the D matrix contains negative values it is also possible to automatically set these to zero using "negatives_to_zero = True". This is not advised as it is better to either keep these negative values, or to remove them using a baseline shift or other pre-processing tool.

Loading pure component absorbance data (S)

It is also possible to input S matrices. These can also be inputted in the form of CSV or txt files. In the example provided, "Ad_4_sdae_sim.py" a model is simulated given pure component absorption data. For CSVs the components label the columns and the wavelengths label the rows. The relevant absorption values fill the matrix. For text files the rows can be unordered, but must be column 1: wavelength, column 2: component name, column 3: absorption. To input this into KIPET we use a similar format as previously described:

```
S_frame = read_absorption_data_from_txt(filename)
```

After this is formatted into the Pandas DataFrame using the above code, we will need to add the data to our model using:

```
builder.add_absorption_data(S_frame)
```

If we plan to use this data to simulate a specific system and perhaps generate a data file (spectra) we will also need to add measurement times to our new model. In the example this is chosen as:

```
builder.add_measurement_times([i*0.0333 for i in range(300)])
```

Loading concentration data (C)

If we wish to do parameter estimation for a problem where we have concentrations that are measured directly by laboratory instruments, this is possible within KIPET, as described in section 4.8 of this document. We can read data in either CSV or txt format using either:

```
C_frame = read_concentration_data_from_txt(filename)
```

or

```
C_frame = read_concentration_data_from_csv(filename)
```

5.1.2 Generating input matrices

It is also possible to generate matrices using some of the built-in functions in KIPET. One such function is able to generate pure-component absorbance data based on Lorentzian parameters:

```
S_frame = generate_absorbance_data(wl_span,S_parameters)
```

Where wl_span is the wavelength span that you wish to generate the data for in the form of a vector where the first entry is the starting wavelength, second is the ending wavelength and the third is the step-length. S_parameters is a dictionary of parameters for the Lorentzian distribution function, ‘alphas’, ‘betas’, and ‘gammas’. This function will generate an absorbance profile DataFrame.

It is also possible to generate a random absorbance data with the function:

```
generate_random_absorbance_data(wl_span,component_peaks, component_widths =
                                None, seed=None)
```

Where the wl_span is the same as above, component_peaks is the maximum height of the absorbance peak, component_widths is the maximum Lorentzian parameter for gamma, and seed is the possible seed for the random number generator.

5.1.3 Writing matrices to files

KIPET also provides functions to write generated matrices to a file with the following functions:

```
write_absorption_data_to_csv(filename,dataframe)
write_absorption_data_to_txt(filename,dataframe)
```

Where the user can define the filename and which DataFrame to input.

It is also possible that a user might wish to generate a spectral D-matrix from a file and then write this to a file. This can be done using:

```
write_spectral_data_to_txt(filename,dataframe)
write_spectral_data_to_csv(filename,dataframe)
```

Which takes in the same inputs as the other function, filename in the form of a string to be used as the output file name and dataframe which is the pandas DataFrame that is to be written.

The same data_tools exist for C-matrices:

```
write_concentration_data_to_txt(filename,dataframe)
write_concentration_data_to_csv(filename,dataframe)
```

5.1.4 Plot spectral data

It is possible to directly plot spectral data using:

```
plot_spectral_data(dataFrame,dimension='2D')
```

Where the inputs dataFrame is the spectral data matrix that you wish to plot and the dimension is the dimension of the graph. For the D-matrix, it is more appropriate to change the dimension to ‘3D’ in order to plot the spectra with time as well as wavelength and absorbance.

5.1.5 Multiplicative Scatter Correction (MSC)

If the experimental measurement data obtained suffers from the scaling or offset effects commonly experienced in spectroscopic measurements, then Multiplicative Scatter Correction (MSC) can be used to pre-process the data using the following function.

```
D_frame = read_spectral_data_from_txt(filename)
mD_frame = msc(dataFrame = D_frame)
```

Automatically the reference spectra is assumed to be the average of each spectrum at each time period. If the user wishes to use a different reference spectrum it can be inputted as a pandas dataframe using the argument reference_spectra=dataframe. An example where MSC is used prior to a Savitzky-Golay filter is provided in Ex_2_estimation_filter_msc.py.

5.1.6 Standard Normal Variate (SNV)

If the experimental measurement data obtained suffers from the scatter effects commonly experienced in spectroscopic measurements, then Standard Normal Variate (SNV) can be used to pre-process the data with:

```
D_frame = read_spectral_data_from_txt(filename)
sD_frame = snv(dataFrame = D_frame)
```

SNV is a weighted normalization method that can be sensitive to very noisy entries in the spectra, so it is possible that SNV increases nonlinear behaviour between the S and C matrices, especially as it is not a linear transformation. An additional user-provided offset can be applied to avoid over-normalization in samples that have near-zero standard deviation. The default value is zero, however this could be improved through applying an offset of close to the expected noise level value through the argument offset= noise. The example Ex_2_estimation_filter_snv.py shows this technique applied to an example prior to Savitzky-Golay filtering.

5.1.7 Savitzky-Golay filter

The Savitzky-Golay (SG) filter is used for smoothing noise from data, with the option to also differentiate the data. It does this by creating a least-squares polynomial fit within successive time windows. In order to implement this smoothing pre=processing step in KIPET the following function is called:

```
fD_frame = savitzky_golay(dataFrame = sD_frame, window_size = 15,orderPoly = 2)
```

Where the user needs to provide the Pandas DataFrame to be smoothed, the number of points over which to apply each smoothing function (window_size) as well as the order of the polynomial to be fitted. Low order polynomials can aggressively smooth data. SNV is commonly employed prior to smoothing to remove scatter effects and an example of this is found in Ex_2_estimation_filter_snv.py.

A further optional option is to differentiate the data as well, using the orderDeriv argument. This option results in the entire KIPET formulation changing to allow for negative values in the D and S matrices. This option may result in longer solve times and strange-looking solutions as allowing for non-negativity constraints to be relaxed, the rotational ambiguity is increased. An example of this is demonstrated in Ex_2_estimation_filter_deriv.py.

5.1.8 Baseline Shift

If the data matrix contains negative values or has a known shift, then we can implement a baseline shift (adding or subtracting a value from all the data):

```
D_frame = read_spectral_data_from_txt(filename)
baseline_shift(dataFrame, shift=None)
```

If shift is not inputted then the function automatically detects the lowest value in the data and shifts the matrix up or down so that this value is zero. This automatically removes negative values from the dataset. If a specific shift is inputted then the data is shifted by that numerical value.

5.1.9 Adding normally distributed noise

In some simulation cases it may be necessary to add noise to simulated data in order to use this to test estimability or parameter estimation functions. It is possible to use the data_tools function in the following way:

```
data = add_noise_to_signal(data, size)
```

This function ensures that Gaussian noise is added to the data (a dataframe) of size (int). The function ensures that no negative values are included by rounding up any negative numbers to 0.

5.2 Pyomo Simulator

While tutorial 1 already explained how to use the simulator class, for completion this section provides the full array of options available to the user for the run_sim function:

```
run_sim(solver,**kwds):
    """ Runs simulation by solving a nonlinear system with ipopt

    Arguments:
        solver (str, required): name of the nonlinear solver to used

        solver_opts (dict, optional): Options passed to the nonlinear
        solver.
        variances (dict, optional): Map of component name to noise
        variance. The map also contains the device noise variance.

        tee (bool,optional): flag to tell the simulator whether to stream
        output to the terminal or not

    Returns:            None
```

5.3 Optimizer Class

Since tutorial 2 already explains how to make use of the functions in this section, for completion the user is provided with the full array of options available to the user for the run_lsq_given_P function which can be used to initialize any of the optimization functions to obtain parameters or variances:

```
run_lsq_given_P(self,solver,parameters,**kwds):
        """Gives a raw estimate of S given kinetic parameters based on a
  difference of least-squares analysis
  Arguments:
      solver (str, required): name of the nonlinear solver to used

      solver_opts (dict, optional): options passed to the nonlinear solver

       variances (dict, optional): map of component name to noise variance.
                                   The map also contains the device noise
                                   variance

         tee (bool,optional): flag to tell the optimizer whether to stream
                              output to the terminal or not

 initialization (bool, optional): flag indicating whether result should be
                                   loaded to the pyomo model or not

  Returns:  Results object with loaded results
```

5.4 VarianceEstimator

Since tutorial 2 already explains how to make use of the functions in this section, for completion the user is provided with the full array of options available to the user for the run_opt function for the VarianceEstimator class:

```
def run_opt(self, solver, **kwds):

        """Solves variance estimation problem following the procedure shown in
           Figure 4 of the documentation. This method solved a sequence of
           optimization problems to determine variances and also automatically
           sets initialization for the parameter estimation for the variables.

        Args:
            solver_opts (dict, optional): options passed to the nonlinear
            solver

            tee (bool,optional): flag to tell the optimizer whether to stream
            output to the terminal or not.

            norm (optional): norm for checking convergence. The default value is
                             the infinity norm (np.inf), it uses same options
                             as scipy.linalg.norm

            report_time (optional, bool): True if we want to report the time
                                  taken to run the variance estimation

            max_iter (int,optional): maximum number of iterations for the
                                  iterative procedure. Default 400.
            tolerance (float,optional): Tolerance for termination by the change
                                        Z. Default 5.0e-5
            subset_lambdas (array_like,optional): Subset of wavelengths to used
                                              for the initialization problem,
                                              as described in Chen, et al.
                                              (2016). Default all wavelengths.

            lsq_ipopt (bool,optional): Determines whether to use ipopt for
                                       solving the least squares problems in
                                       the Chen, et al. (2016) procedure.
                                       Default False. The default uses
                                       scipy.least_squares.

            init_C (DataFrame,optional): Dataframe with concentration data used
                                       to start the iterative procedure.
fixed_device_variance (float, optional): if the device variance is
                known ahead of time and you would not like to
                estimate it, set the variance here.
            Returns:  None
```

Note here that the standard method is to use Scipy least squares, which is actually a slower method for the estimation. Additionally, if device variance is known ahead of time from the manufacturer, we are able to input it directly here.

5.5 Parameter Estimator

Since tutorial 2 already explains how to make use of the function in this section, for completion the user is provided with the full array of options available to the user for the run_opt function for the ParameterEstimator class:

```
    def run_opt(self, solver, **kwds):
        """ Solves parameter estimation problem.
        Arguments:
           solver (str): name of the nonlinear solver to used

          solver_opts (dict, optional): options passed to the nonlinear solver

          variances (dict, optional): map of component name to noise variance.
                                      The map also contains the device noise
                                      variance.
          tee (bool,optional): flag to tell the optimizer whether to stream
                               output to the terminal or not.

          with_d_vars (bool,optional): flag to the optimizer whether to add
                                  variables and constraints for D_bar(i,j),
                                  which is included when we have a problem
                                  with noise
            report_time (optional, bool): True if we want to report the time
                                  taken to run the parameter estimation
          covariance(bool, optional): if this is selected, the confidence
                                  intervals will be calculated for the
                                  estimated parameters. If this is selected
                                  then the solver to be used should be
                                  'ipopt_sens' or 'k_aug' or else an error
                                  will be encountered.
        Returns:  Results object with loaded results
```

5.6 Troubleshooting and advanced strategies for difficult problems

Since the problems that KIPET is solving are often highly non-linear and non-convex NLPs, it is often important to provide the solver (IPOPT) with good initial points. This section will briefly describe some of the additional initialization and solver strategies that can be applied in KIPET in order to solve larger and more difficult problems. This section assumes that the user has read the tutorial problems above.

Since the VarianceEstimator needs to solve the full optimization problem, it may be useful to initialize it. It is possible to do this by fixing the unknown parameters to some value (hopefully fairly close to the real values) and then running a least squares optimization in order to get decent initial values for the variables, Z, S, dZ/dt, and C. eg.. KIPET provides the ability to do this through an easy to implement function:

```
    p_guess = {'k1':4.0,'k2':2.0}
    raw_results = v_estimator.run_lsq_given_P('ipopt',p_guess,tee=False)
v_estimator.initialize_from_trajectory('Z',raw_results.Z)
v_estimator.initialize_from_trajectory('S',raw_results.S)
    v_estimator.initialize_from_trajectory('dZdt',raw_results.dZdt)
```

```
v_estimator.initialize_from_trajectory('C',raw_results.C)
```

This will allow the user to initialize the VarianceEstimator using the same methods and functions described in the tutorial sections. Note that it is possible to use the run_lsq_given_P() to initialize the ParameterEstimator method as well if the variances are known or there is no need to compute variances. An example of this is shown in Ad_1_estimation.py.

When running the ParameterEstimator it is possible to improve solution times or to assist IPOPT in converging to a solution by not only providing initializations (either through a least squares with fixed parameters or using the results from the VarianceEstimator) as shown above but also by scaling the NLP. KIPET provides a tool to provide automatic scaling based on the VarianceEstimator's solution with the following function.

```
p_estimator.scale_variables_from_trajectory('Z',results_variances.Z)
p_estimator.scale_variables_from_trajectory('S',results_variances.S)
p_estimator.scale_variables_from_trajectory('C',results_variances.C)
```

and this can then be given to the solver as an option in the following way:

```
options = dict()
options['nlp_scaling_method'] = 'user-scaling'
results_pyomo = p_estimator.run_opt('ipopt',  tee=True, solver_opts =
                                    options,variances=sigmas, with_d_vars=True)
```

If convergences are extremely slow it is also possible to provide the solver with an additional option that changes the barrier update strategy. This option may not necessarily be required, but can help with some problems, especially with noisy data. This is added to the solver options with this:

```
solver_options['mu_strategy'] = 'adaptive'
```

Another useful solver option that has not yet been mentioned in this guide and which might help to improve the chances of obtaining a solution is the:

```
options['bound_push'] =1e-6
```

Which is the desired minimum distance from the initial point to bound. By keeping this value small it is possible to determine how much the initial point might have to be modified in order to be sufficiently within the bounds.

More information on the IPOPT solver and the available options can be found here:

https://www.coin-or.org/Ipopt/documentation/node2.html

In some cases it can be useful to give initial values for the parameters solving the parameter estimation problems. This can be done providing an additional argument named init, e.g.

builder.add_parameter('k1',init=1.0,bounds=(0.0,10.0))

An example can be found in Ex_2_estimationfefactoryTempV.py.

Furthermore, it might be useful to provide nonnegative bounds for algebraic variables for example for rate laws. To achieve this, add the ones, here r1, with bounds to the TemplateBuilder in the following way

```
builder.add_algebraic_variable('r1', bounds=(0.0, None))
```

instead of adding them as a set. This might be useful in some cases but it also restricts the optimization algorithm in a higher manner, such that it can be more difficult to find a solution.

Another particularly useful feature of KIPET is that we can set certain profiles to have specific features or bounds. And example of this is if we know that some peak exists on one of the pure components' absorbance or if we know that a certain species' concentration never exceeds a certain number. To implement bounds such as these, we can use the function:

```
builder.bound_profile(var = 'S', comp = 'A', bounds = (50,65), profile_range =
                                                    (1650,1800))
```

Here the var is which of the profiles we want to bound, comp is the component/species, bounds are the bounds that we wish to impose and profile_range is the specific area we wish to impose the bound. In this case, species A's absorbance is bounded to between 50 and 65 in the wavelength range of 1650 to 1800. More examples of this are included in the example Ex_2_estimation_bound_prof_fixed_variance.py.

With problems that are difficult to solve it can also be useful to not just initialize the primal variables but also the dual variables from a previous solution. For this the following options should be provided:

```
options['warm_start_init_point']='yes'
options['warm_start_bound_push'] = 1e-9
options['warm_start_mult_bound_push'] = 1e-9
options['mu_strategy']='adaptive'
```

and the warmstart argument should be set to true:

```
results_pyomo = p_estimator.run_opt('ipopt',
                                    tee=True,
                                    solver_opts=options,
                                    variances=sigmas,
                                    with_d_vars=True,
                                    warmstart=True)
```

An example is provided in Ad_2_estimation_warmstart.py, where we just estimate one parameter first and then initialize the estimation of both parameters with that solution.

6. REFERENCES

Chen, W., Biegler, L.T., Garcia-Munoz, S., 2016, An Approach for Simultaneous Estimation of Reaction Kinetics and Curve Resolution from Process and Spectral Data, Journal of Chemometrics, 30, 506-522.

Fogler, H. S., 2006, Elements of Chemical Reaction Engineering (4th Edition), Prentice Hall International.

Hart, W.E., Laird, C., Watson, J.P., Woodruff, D.L., 2012, Pyomo: Optimization Modeling in Python, volume 67, Springer Verlag.

HSL, 2013, A collection of Fortran codes for large scale scientific computation. http://www.hsl.rl.ac.uk

Hunter, J.D., 2007, Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95, DOI:10.1109/MCSE.2007.55

Jaumot, J., Gargalloa, R., de Juana, A., Tauler, R., 2005, A graphical user-friendly interface for MCR-ALS: a new tool for multivariate curve resolution in MATLAB. Chemometrics and Intelligent Laboratory Systems, 76, 101–110

McKinney W., 2013, Python for Data Analysis: Data Wrangling with Pandas, NumPy, and Ipython

Puxty, G., Maeder, M., Hungerbuhler, K., 2006, Tutorial on the fitting of kinetics models to multivariate spectroscopic measurements with non-linear least-squares regression. Chemometrics and Intelligent Laboratory Systems, 81, 149–164.

Sawall, M., Boerner, A., Kubis, C., Selent, D., Ludwig, R., Neymeyr, K., 2012, Model-free multivariate curve resolution combined with model-based kinetics: algorithm and applications. J. Chemometrics, 26: 538–548.

van der Walt, S., Colbert, C., Varoquaux, G., 2011, The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30, DOI:10.1109/MCSE.2011.37

Wächter, A., Biegler, L.T., 2006, On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming, Mathematical Programming, 106(1): 25-57.

Wu, S., McLean, K. A. P., Harris, T. J., McAuley, K. B., 2011, Selection of optimal parameter set using estimability analysis and MSE-based model-selection criterion. Int. J. Advanced Mechatronic Systems, 3(3), 188-197.

Yao, K.Z., Shaw, B.M., McAuley, K.B., Bacon, D.W., 2003, Modeling ethylene/butene copolymerization with multi-site catalysts: parameter estimability analysis and experimental design, Polym. React. Eng., 11 (3), 563-588.