<table>
<thead>
<tr>
<th>1</th>
<th>Table of Contents</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Installation Guide</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Overview/Introduction</td>
<td>9</td>
</tr>
<tr>
<td>1.3</td>
<td>Citing KIPET</td>
<td>11</td>
</tr>
<tr>
<td>1.4</td>
<td>Background</td>
<td>11</td>
</tr>
<tr>
<td>1.5</td>
<td>Tutorials</td>
<td>19</td>
</tr>
<tr>
<td>1.6</td>
<td>Additional Functions</td>
<td>58</td>
</tr>
<tr>
<td>1.7</td>
<td>References</td>
<td>67</td>
</tr>
<tr>
<td>2</td>
<td>Indices and tables</td>
<td>69</td>
</tr>
<tr>
<td>3</td>
<td>KIPET Resources</td>
<td>71</td>
</tr>
</tbody>
</table>
KIPET is the one-stop shop for kinetic parameter estimation from batch and fed-batch reactor systems using spectroscopic or concentration data. KIPET is a Python-based package using maximum-likelihood statistics, large-scale nonlinear programming optimization, and finite element discretization in a unified framework to solve a variety of parameter estimation problems. Use KIPET to:

- Simulate reactive system described with DAEs
- Solve DAE systems with collocation methods
- Pre-process data
- Perform estimability analysis
- Estimate data variances
- Estimate kinetic parameters
- Estimate confidence intervals of the estimated parameters
- Estimate parameters from multiple datasets with different experimental conditions
- Obtain the most informative wavelength set to obtain minimal lack-of-fit
- Analyze your system (SVD, PCA, lack of fit, etc.)
- Visualize results
1.1 Installation Guide

This section will explain how to install KIPET on different operating systems. Since this guide is meant to provide an introduction to all kinds of users, it shall start with the basics. For those with experience in Python, proceed to the later sections in this chapter. Firstly Python will need to be installed.

1.1.1 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.

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 anaconda

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.

**MacOS installation**

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”.

**Linux installation**

After downloading Anaconda 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.

**1.1.2 Installing Packages and Dependencies**

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

**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.

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

A complete list of all the dependencies is included in Table Table 1.1.

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:

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

or via the Anaconda Prompt:

```bash
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:

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

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

```bash
conda install -c conda-forge pyomo==5.6.1
```

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:

```bash
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 1.1: List of dependencies for KIPET

<table>
<thead>
<tr>
<th>Package</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>appdirs</td>
<td>1.4.3</td>
</tr>
<tr>
<td>backports.functools-lru-cache</td>
<td>1.5</td>
</tr>
<tr>
<td>casadi</td>
<td>3.4.0</td>
</tr>
<tr>
<td>coverage</td>
<td>4.5.1</td>
</tr>
<tr>
<td>cycler</td>
<td>0.10.0</td>
</tr>
<tr>
<td>decorator</td>
<td>4.2.1</td>
</tr>
<tr>
<td>kiwisolver</td>
<td>1.0.1</td>
</tr>
<tr>
<td>matplotlib</td>
<td>2.2.0</td>
</tr>
<tr>
<td>networkx</td>
<td>2.1</td>
</tr>
<tr>
<td>nose</td>
<td>1.3.7</td>
</tr>
<tr>
<td>numpy</td>
<td>1.14.2</td>
</tr>
<tr>
<td>pandas</td>
<td>4.5.1</td>
</tr>
<tr>
<td>ply</td>
<td>3.11</td>
</tr>
<tr>
<td>pyomo</td>
<td>5.6.1</td>
</tr>
<tr>
<td>pyparsing</td>
<td>2.2.0</td>
</tr>
<tr>
<td>Python-dateutil</td>
<td>2.7.0</td>
</tr>
<tr>
<td>pytz</td>
<td>2018.3</td>
</tr>
<tr>
<td>scipy</td>
<td>1.0.0</td>
</tr>
<tr>
<td>PyUtilib</td>
<td>5.6.5</td>
</tr>
<tr>
<td>six</td>
<td>1.11.0</td>
</tr>
</tbody>
</table>

#### 1.1.3 Installing KIPET

Firstly, KIPET’s source code can be downloaded from [https://github.com/salvadorgarciamunoz/kipet.git](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:
1.1.4 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:
Run an example (explained in the next section) to test if it works. This method should also include a functioning version of slpopt 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.

### 1.1.5 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](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.

### 1.1.6 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](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 slpopt as no binaries for slpopt 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”.

### 1.1.7 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!

### 1.1.8 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`. 

---

**Chapter 1. Table of Contents**

1. Installation
   - 1.1 Installing KIPET
      - 1.1.1 Introduction
      - 1.1.2 Installation Requirements
      - 1.1.3 Installation Instructions
      - 1.1.4 Installation Tools
      - 1.1.5 Installing k_aug
      - 1.1.6 sIPOPT installation
      - 1.1.7 Windows PATH Management
      - 1.1.8 Validation of the Package

2. Validation
3. Conclusion
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.

1.1.9 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.

1.2 Overview/Introduction

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 do estimability analysis. This is all possible through a unified framework based on maximum likelihood principles, robust discretization methods, and large-scale nonlinear optimization. In this documentation the capabilities of KIPET are described and examples and tutorials are given so that a user with basic programming knowledge can use the toolkit for their own purposes.

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
- Pre-process data
- Estimability analysis
- Estimate data variances

Fig. 1.1: Visualization of a multi-wavelength spectroscopic dataset
Fig. 1.2: Example of the decomposition of the spectra in single component absorbance and concentration profiles (absorbance)

Fig. 1.3: Example of the decomposition of the spectra in single component absorbance and concentration profiles (concentration)
• Estimate kinetic parameters
• 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.

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. The KIPET team involves Salvador García-Munoz (Eli Lilly), Santiago Rodriguez (Purdue University), Christina Schenk (Basque Center for Applied Mathematics), Michael Short (University of Surrey), Lorenz T. Biegler, David M. Thierry, Kevin McBride, and Kuan-Han Lin (all Carnegie Mellon University).

When using KIPET please cite:


1.3 Citing KIPET

Please use the following when citing KIPET:


1.4 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
1.4.1 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

This can be done for problems where we have multiple datasets from separate experiments or in the case of having concentration data only and not spectra.

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.

![Fig. 1.4: The steps/modules involved in setting up a KIPET model](image-url)
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 and necessary for problems in which we have dosing). 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. We provide a number of different approaches to estimate the variances. The first one is the one described 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:

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 the next figure.

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. The above method was described in the initial paper from Chen et al. (2016). This method can be problematic for certain problems and so a new variance estimation procedure has been developed and implemented in version 1.1.01 whereby direct maximum likelihood formulations are solved. We propose and include 2 new methods as well as a number of functions in order to obtain good initial guesses for variance. The first and recommended method is known as the “alternate” strategy within KIPET. Here we solve for the worst-case device variance first:
Fig. 1.6: Variance estimation equations

\[
\frac{1}{n_t} \sum_{i=0}^{n_t} \left( D_{i,l} - \sum_{k=0}^{n_c} z_k(t_i)s_k(\lambda_l) \right)^2 = \sum_{k=0}^{n_c} s_k(\lambda_l)\sigma_k^2 + \delta^2
\]

\[\forall l = 1, \ldots, n_w\]

\[Z, S, \delta^2, \sigma_k^2\]

\[\min \log \left( \sum_i \tilde{\epsilon}_i^T \tilde{\epsilon}_i \right)\]
where

\[
\tilde{\epsilon}_{ij} = d_{ij} - \sum_k z_k(t_i)s_{kj}
\]

Then we set:

\[
v^2 = \sum_i \tilde{\epsilon}_i^T \tilde{\epsilon}_i / (nwp ntp)
\]

We also know that, from derivations in Chen et al. (2016):

\[
v^2 = \sum_{l=1}^{nwp} \sum_{k=1}^{nc} (\delta^2 + \sigma^2 s_{kl}) / nwp.
\]

We guess initial values for \(\sigma\) (which the user provides) and solve the maximum likelihood-derived objective:

and then we are able to determine delta from:

Following this we can evaluate:

This function then provides us with the difference between our overall variance and the model and device variances.

If the value of the function is below tolerance we stop or we update \(\sigma\) using a secant method and re-solve until we find convergence.

A third method is provided, referred to as “direct_sigmas” in KIPET, which first assumes that there is no model variance and solves directly for a worst-case device variance. The formulation solved is thus:

And from this formulation, the device variance can be solved for directly assuming that there is no model variance.

Once the worst-possible device variance is known, we can obtain some region in which to solve for the model variances knowing the range in which the device variance is likely to lie. The following equation can be used to solve for different values of device variance:

Once these solutions are obtained we can solve directly for the model variances. A selection of model and device variances are then provided to the user, and the user is able to decide on the appropriate combination for their problem.

More rigorous mathematical derivations of these methods will be provided in future documentation versions. 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:

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 or k_aug to perform the optimization with sensitivities in order to determine the confidence intervals.
\[
\min nwp \log \left( \sum_i \epsilon_i^T \epsilon_i \right) + \sum_i \frac{\eta_i^T \eta_i}{2\sigma^2}
\]

\[
\delta^2 = \sum_i \epsilon_i^T \epsilon_i / (nwp \ ntp)
\]

\[
f(\sigma_p) = \nu^2 - \delta^2 - \sigma_p^2 \left( \sum_{l=1}^{nwp} \sum_{k=1}^{nc} s_{kl} / nwp \right)
\]

\[
\min \frac{nwp \ ntp}{2} \log \left( \sum_{i=1}^{nwp} \sum_{l=1}^{nwp} (d_{i,l} - \sum_{k=1}^{nc} z_k (t_i) s_k (\lambda_i))^2 / (ntp \ nwp) \right)
\]

\[
\min \frac{1}{2\delta^2} \sum_{i=1}^{ntp} \sum_{l=1}^{nwp} (d_{i,l} - \sum_{k=1}^{nc} z_k (t_i) s_k (\lambda_i))^2 + \frac{ntp \ nc}{2} \log \left( \sum_{i=1}^{nwp} \sum_{k=1}^{nc} (c_k (t_i) - z_k (t_i))^2 / (ntp \ nc) \right)
\]

\[
\min \sum_{i=1}^{ntp} \sum_{l=1}^{nc} (d_{i,l} - \sum_{k=1}^{nc} c_k (t_i) s_k (\lambda_i))^2 / \delta^2 + \sum_{i=1}^{ntp} \sum_{k=1}^{nc} (c_k (t_i) - z_k (t_i))^2 / \sigma_k^2
\]

s.t. \quad \sum_{m=0}^{K} \sum_{m=0}^{K} \sum_{m=0}^{K} \sum_{m=0}^{K} l_m (\tau) z_{jm} - h_j f (z_{jm}, \theta) = 0, j = 1..nc, m = 1..K

\[
z^K (t_i) = \sum_{m=0}^{K} l_m (\tau) z_{jm}, \tau = (t_i - tp_{j-1}) / (tp_j - tp_{j-1})
\]

\[
C \geq 0, S \geq 0
\]
1.4.2 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.

1.4.3 Modeling approach and organization

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.
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:

**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.

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

**TemplateBuilder**

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

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

**Simulator**

This section is required if a simulation is being performed and can call either upon PyomoSimulator or 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.

```python
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 *

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 *

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 *

MultipleExperimentsEstimator

Class called when performing parameter estimation from multiple datasets. Works through constructing pyomo blocks containing each individual dataset and performing the variance and parameter estimation separately before solving them simultaneously with parameters linked across datasets.

from kipet.library.MultipleExperimentsEstimator import *

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 parameters fixed and others remaining as variables. Currently only functional for concentration-data problems.

from kipet.library.EstimabilityAnalyzer import *

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.

1.5 Tutorials

In this section we show a collection of tutorials, all of which are included in the examples folder in KIPET’s main directory. In addition to the tutorial examples that are covered here which show a wide range of KIPET’s functionality, there are extra examples that are listed in the table at the end of this section.
1.5.1 A Note About the New Version

If you have previously used KIPET, then this tutorial will look somewhat familiar, but also very different. There are many new changes to the structure of KIPET that greatly reduce the amount of coding on the part of the user. Whereas before you had to create instances of the VarianceEstimator, the ParameterEstimator, or of other classes, this is no longer required and has for the most part been automated. You can still modify the behavior of these tools through the new settings attribute of the top-level KipetModel object or the ReactionModel objects. These can be seen in the updated tutorial exercises that follow.

1.5.2 Tutorial 1 – Simulating a Simple Example

Files Ex_1_ode_sim.py

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.

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

Before going into more detail, the complete block of code required to simulate this simple reaction is presented. As you can see, the user does not require much coding to use KIPET.

```python
from kipet import KipetModel
kipet_model = KipetModel()

r1 = kipet_model.new_reaction('reaction-1')

# Add the model parameters
r1.add_parameter('k1', 2)
r1.add_parameter('k2', 0.2)

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=1)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0)

# Define explicit system of ODEs
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

# Add the rules to the model
r1.add_equations(rule_odes)

# Create the model - simulations require times
r1.set_times(0, 10)

# Simulate with default options
r1.simulate()

r1.results.plot()
```
We will now break this down step by step. The first step is to import the kipet module or the KipetModel class from the kipet module as in the example.

```python
from kipet import KipetModel
```

The KipetModel class contains all of the methods necessary to use KIPET. The next step is to create an instance of the KipetModel class.

```python
kipet_model = KipetModel()
```

The KipetModel class has the attribute “models” that acts like a dictionary and contains instances of the ReactionModel, which is where individual reaction models are contained. To create a new reaction model, it is added to the KipetModel using the method “new_reaction”. Note that the reaction requires a name as the first argument.

```python
r1 = kipet_model.new_reaction('reaction-1')
```

We can now use the ReactionModel instance “r1” to add all of the expected model components such as the kinetic model and its parameters, the component information, and the data (if any). Parameters are added using the method “add_parameter”, as seen in the current example where there are two parameters:

```python
r1.add_parameter('k1', 2)
r1.add_parameter('k2', 0.2)
```

Since our system has three components, A, B, and C, these need to be declared as well. Each component requires a name and a state. The state can be concentration, spectral, state, trajectory, or custom. Each one of these states will be covered in more detail later in this tutorial. At the moment, the only state we are concerned about is concentration. For simulations, an initial value for each of the components is required.

```python
r1.add_component('A', state='concentration', init=1)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0)
```

The next step is to provide the reaction kinetics in a Pyomo friendly format. The reaction scheme for this example is defined as a python function containing each component’s specific ODE within a dictionary. This is probably the most coding work that will be necessary when using KIPET. The reaction kinetic rules are placed into the model using the “add_equations” method. Please note that KIPET requires that each declared component has its own expression.

```python
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
r1.add_equations(rule_odes)
```

At this point we have provided KIPET with a reaction model, component information, and parameter data. The only thing still required for our simulation are the start and end times. Times are provided using the method “set_times”.

```python
r1.set_times(0, 10)
```

After this we are ready to simulate using the “simulate” method. The results are then accessible using the “results” attribute. This is an instance of the ResultsObject class, which contains convenient plotting methods to display the results. The most basic plotting tool can be accessed using the “plot” method of the “results” attribute.
r1.simulate()
r1.results.plot()

The results are then presented in a new browser tab using Plotly similar to the following figure.

![content/ex_1_plot_sim_results.png](content/ex_1_plot_sim_results.png)

Fig. 1.7: Plot obtained from tutorial example 1

### 1.5.3 Tutorial 2 – Parameter Estimation

**Files** Ex_2_estimation.py

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 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.

The full code for this example:

```python
from kipet import KipetModel
kipet_model = KipetModel()

r1 = kipet_model.new_reaction('reaction-1')

# Add the model parameters
r1.add_parameter('k1', init=2, bounds=(0.0, 5.0))
r1.add_parameter('k2', init=0.2, bounds=(0.0, 2.0))

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=1)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0)

# Add data to model
r1.add_dataset('D_frame', category='spectral', file='Dij.txt')

# define explicit system of ODEs
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
r1.add_equations(rule_odes)

r1.bound_profile(var='S', bounds=(0, 200))

# Settings
r1.settings.collocation.ncp = 1
r1.settings.collocation.nfe = 60
r1.settings.variance_estimator.use_subset_lambdas = True
r1.settings.parameter_estimator.tee = False
r1.settings.parameter_estimator.solver = 'ipopt_sens'

# Solve and display the results
r1.run_opt()
r1.results.show_parameters
r1.results.plot()
```

As you can see, much of the modeling is similar to those found in Tutorial 1. The differences between the two are explained below.

**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 “add_dataset” method. If you are using spectral data, as in this case, the “category” argument needs to be ‘spectral’. The location of the file should be given with the “file” argument.
r1.add_dataset(category='spectral', file='Dij.txt')

The KipetModel instance has a default data directory where it expects the data file to be found. This can be changed to another directory if desired through the settings object:

```python
kipet_model.settings.general.data_directory = 'path/to/data/directory'
```

KIPET has many built-in features to handle the data after adding it to the model. These features are explored in later tutorials.

Also new here is the optional declaration of bounds for the individual species absorbance (S) profiles. This is done using the “bound_profile” method:

```python
r1.bound_profile(var='S', bounds=(0, 200)
```

### Settings

If you wish to change the default settings for the collocation method, you can access these using the settings attribute. The code below shows how to change the number of collocation points (ncp) and the number of finite elements (nfe) for the variance estimation and the parameter estimation.

```python
r1.settings.collocation.ncp = 1
r1.settings.collocation.nfe = 60
```

You can also limit the set of wavelengths to use in initializing the problem. For large problems it might be worthwhile using smaller wavelength subsets to work with less data, saving computational time. 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. This can be accessed using the “use_subset_lambdas” option by setting it equal to True. You can choose the frequency of wavelengths by changing the freq_subset_lambdas option to an integer number from the default of 4.

```python
r1.settings.variance_estimator.use_subset_lambdas = True
# Default
r1.settings.variance_estimator.freq_subset_lambdas = 4
```

Many of the required options for the variance estimation and parameter estimation can be accessed using the settings attribute of the ReactionModel instance. You can look at the various options by printing the settings attribute to the console.

```python
print(r1.settings)
```

You can also change whether or not the optimization results are displayed or not (tee) and the solver used to fit the parameters (solver). In this case, you can use ipopt_sens to calculate the confidence intervals for the parameters.

```python
r1.settings.parameter_estimator.tee = False
r1.settings.parameter_estimator.solver = 'ipopt_sens'
```

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.
Solver settings can be set in the following manner:

```
r1.settings.solver.<solver setting> = <option>
```

**Variance Estimation and Parameter Fitting**

To solve the problem, simply run the following:

```
r1.run_opt()
```

The results and plots can now be displayed.

```
r1.results.show_parameters
r1.results.plot()
```

The results will then be shown as:

```
The estimated parameters are:
k2 0.201735984306
k1 2.03870135529
```

Providing us with the following plots:

**1.5.4 Tutorial 3 – Advanced reaction systems with additional states**

**Files**  Ex_3_complementary.py

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 the complementary state variable temperature is added as a component using the state “state”.

```
# This is needed for the construction of the ODEs
from pyomo.core import exp

from kipet import KipetModel
kipet_model = KipetModel()

r1 = kipet_model.new_reaction('reaction-1')

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=1.0)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0)

# Complementary states have the state 'state'
r1.add_component('T', state='state', init=290)
r1.add_component('V', state='state', init=100)
```

When using complimentary states (not concentration), the variables are labeled using “X” instead of “Z”. 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:
Fig. 1.8: Concentration profile results from tutorial example 2
Fig. 1.9: Pure component absorbance profiles (S) result from tutorial example 2
# Define the ODEs

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

r1.add_equations(rule_odes)
r1.set_times(0.0, 2.0)
r1.settings.collocation.nfe = 20
r1.settings.collocation.ncp = 1
r1.simulate()
r1.results.plot()

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.
1.5.5 Tutorial 4 – Simulation of Advanced Reaction system with Algebraic equations

Files  Ex_4_sim_aspirin.py

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:

\[
\begin{align*}
SA + AA & \underset{k_1}{\rightarrow} ASA + HA \\
ASA + AA & \underset{k_2}{\rightarrow} ASAA + HA \\
ASAA + H_2O & \underset{k_3}{\rightarrow} ASA + HA \\
AA + H_2O & \underset{k_4}{\rightarrow} 2HA \\
SA(s) & \underset{k_d}{\rightarrow} SA(l) \\
ASA(l) & \underset{k_e}{\rightarrow} ASA(s)
\end{align*}
\]

With the rate laws being:

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

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.

```python
import pandas as pd
from pyomo.environ import exp
```

(continues on next page)
\begin{align*}
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 (c_{SA}^{\text{sat}}(T) - c_{SA}(t))^d, & \text{if } m_{SA}(t) \geq 0 \\
    0, & \text{if } m_{SA}(t) < 0 
\end{cases} \\
r_g &= k_c \left( \max(c_{ASA}(t) - c_{ASA}^{\text{sat}}(T), 0) \right)^c 
\end{align*}

\begin{align*}
\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} \\
\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_{H_2O}^{\text{in}} - \frac{\dot{V}}{V} c_{H_2O} \\
\dot{V} &= V \sum_{i=1}^{n_s} \left( \sum_{j=1}^{4} \gamma_{i,j} r_j + \gamma_{i,d} r_d + \gamma_{i,c} r_c + \epsilon \frac{f}{V} c_{H_2O}^{\text{in}} \right)
\end{align*}
from kipet import KipetModel

kipet_model = KipetModel()

r1 = kipet_model.new_reaction('reaction-1')

# Components
components = dict()
components['SA'] = 1.0714 # Salicitilc acid
components['AA'] = 9.3828 # Acetic anhydride
components['ASA'] = 0.0177 # Acetylsalicylic acid
components['HA'] = 0.0177 # Acetic acid
components['ASAA'] = 0.000015 # Acetylsalicylic anhydride
components['H2O'] = 0.0 # water

for comp, init_value in components.items():
    r1.add_component(comp, state='concentration', init=init_value)

# Parameters
params = dict()
params['k0'] = 0.0360309
params['k1'] = 0.1596062
params['k2'] = 6.8032345
params['k3'] = 1.8028763
params['kd'] = 7.1108682
params['kc'] = 0.7566864
params['Csa'] = 2.06269996

for param, init_value in params.items():
    r1.add_parameter(param, init=init_value)

In this example we need to declare new sets of states in addition to our components and parameters, as with Tutorial 3:

# Additional state variables
extra_states = dict()
extra_states['V'] = 0.0202
extra_states['Masa'] = 0.0
extra_states['Msa'] = 9.537

for comp, init_value in extra_states.items():
    r1.add_component(comp, state='state', init=init_value)

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

# Algebraics
algebraics = ['f', 'r0', 'r1', 'r2', 'r3', 'r4', 'r5', 'v_sum', 'Csat']

r1.add_algebraic_variables(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]

(continues on next page)
To define the algebraic equations in Equn (35) we then use:

```python
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)

    # 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
```

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{0}'.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{0}'.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

r1.add_equations(rule_odes)

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:

# Data set-up: Use trajectory as the category for initialization data
# as this is not added to the pyomo model
filename = r1.set_directory('extra_states.txt')
r1.add_dataset('traj', category='trajectory', file=filename)

filename = r1.set_directory('concentrations.txt')
r1.add_dataset('conc', category='trajectory', file=filename)

filename = r1.set_directory('init_Z.csv')
r1.add_dataset('init_Z', category='trajectory', file=filename)

filename = r1.set_directory('init_X.csv')
r1.add_dataset('init_X', category='trajectory', file=filename)

filename = r1.set_directory('init_Y.csv')
r1.add_dataset('init_Y', category='trajectory', file=filename)

where the external files are the csv’s and the state is considered to be “trajectory”. 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).

# Create the model
r1.set_times(0, 210.5257)

# Settings
r1.settings.collocation.nfe = 100
r1.settings.simulator.solver_opts.update({'halt_on_ampl_error': 'yes'})
# If you need to fix a trajectory or initialize, do so here:
```python
r1.fix_from_trajectory('Y', 'Csat', 'traj')
r1.fix_from_trajectory('Y', 'f', 'traj')
```
```python
r1.initialize_from_trajectory('Z', 'init_Z')
r1.initialize_from_trajectory('X', 'init_X')
r1.initialize_from_trajectory('Y', 'init_Y')
```

# Run the simulation
```python
r1.simulate()
```

You can add extra data to the plot method to compare results with other data by using a dictionary as shown below:
```python
# Plot the results
```python
r1.results.plot('Z')
r1.datasets['conc'].show_data()
r1.results.plot('Y', 'Csat', extra_data={'data': r1.datasets['traj'].data['Csat'], 'label': 'traj'})
r1.results.plot('X', 'V', extra_data={'data': r1.datasets['traj'].data['V'], 'label': 'traj'})
r1.results.plot('X', 'Msa', extra_data={'data': r1.datasets['traj'].data['Msa'], 'label': 'traj'})
r1.results.plot('Y', 'f')
r1.results.plot('X', 'Masa', extra_data={'data': r1.datasets['traj'].data['Masa'], 'label': 'traj'})
```

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

Files
- Ex_5_sim_fe_by_fe.py
- Ex_5_sim_fe_by_fe_multiplejumpsandinputs.py

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.

\[
\begin{align*}
AH + B \xrightarrow{k_1} A^- + BH^+ & \quad r_1 = k_{AH} c_B \\
A^- + C \xrightarrow{k_2} AC^- & \quad r_2 = k_2 c_A c_C \\
AC^- \xrightarrow{k_3} A^- + C & \quad r_2 = k_2 c_{AC^-} \\
AC^- + AH \xrightarrow{k_3} P + A^- & \quad r_3 = k_3 c_{AC^-} c_{AH} \\
AC^- + BH^+ \xrightarrow{k_4} P + B & \quad r_4 = k_4 c_{AC^-} c_{BH^-}
\end{align*}
\]

Which is represented by the following ODE system:
```python
import pandas as pd
from pyomo.environ import exp
```
\[
\frac{dV}{dt} = \begin{cases} 
\text{const. flowrate}, & t < 3.5 \text{h} \\
0, & t > 3.5 \text{h}
\end{cases}
\]

\[
\frac{dc_{AH}}{dt} = -r_1 - r_3 - \frac{V}{V} c_{AH}
\]

\[
\frac{dc_B}{dt} = -r_1 + r_4 - \frac{V}{V} c_B
\]

\[
\frac{dc_{A^{-1}}}{dt} = r_1 - r_2 + r_2 - r_3 - \frac{V}{V} c_{A^{-1}}
\]

\[
\frac{dc_{BH^*}}{dt} = -r_1 - r_4 - \frac{V}{V} c_{BH^*}
\]

\[
\frac{dc_C}{dt} = -r_2 + r_2 - \frac{V}{V} c_C + \begin{cases} 
m_{C_{-\text{add}}} / V / 3.5, & t < 3.5 \text{h} \\
0, & t > 3.5 \text{h}
\end{cases}
\]

\[
\frac{dc_{AC^{-1}}}{dt} = r_2 - r_2 - r_3 - r_4 - \frac{V}{V} c_{AC^{-1}}
\]

\[
\frac{dc_p}{dt} = r_3 + r_4 - \frac{V}{V} c_p
\]
from kipet import KipetModel
kipet_model = KipetModel()

rl = kipet_model.new_reaction('simulation')

# components
components = dict()
components['AH'] = 0.395555
components['B'] = 0.0351202
components['C'] = 0.0
components['BH+]'] = 0.0
components['A-'] = 0.0
components['AC-'] = 0.0
components['P'] = 0.0

for comp, init_value in components.items():
    rl.add_component(comp, state='concentration', init=init_value)

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
# note the fifth component. Which basically works as an input
rl.add_algebraic_variables(algebraics)

params = dict()
params['k0'] = 49.7796
params['k1'] = 8.93156
params['k2'] = 1.31765
params['k3'] = 0.310870
params['k4'] = 3.87809

for param, init_value in params.items():
    rl.add_parameter(param, init=init_value)

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

rl.add_component('V', state='state', init=0.0629418)

# stoichiometric coefficients
gammas = dict()
gammas['AH'] = [-1, 0, 0, -1, 0]
gammas['B'] = [-1, 0, 0, 0, 1]
gammas['C'] = [0, -1, 1, 0, 0]
gammas['BH+'] = [1, 0, 0, 0, -1]
gammas['A-'] = [1, -1, 1, 1, 0]
gammas['AC-'] = [0, 1, -1, -1, -1]
gammas['P'] = [0, 0, 0, 1, 1]

def rule_algebraics(m, t):
    r = list()
r.append(m.Y[t, 0] - m.P['k0'] * m.Z[t, 'AH'] * m.Z[t, 'B'])
r.append(m.Y[t, 1] - m.P['k1'] * m.Z[t, 'A-'] * m.Z[t, 'C'])
r.append(m.Y[t, 2] - m.P['k2'] * m.Z[t, 'AC-'])
r.append(m.Y[t, 3] - m.P['k3'] * m.Z[t, 'AC-'] * m.Z[t, 'AH'])
r.append(m.Y[t, 4] - m.P['k4'] * m.Z[t, 'AC-'] * m.Z[t, 'BH+'])
return r
#: there is no AE for Y[t,5] because step eqn under rule_odes functions as the switch for the "C" equation

r1.add_algebraics(rule_algebraics)

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

r1.add_equations(rule_odes)

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.

It is then necessary to declare the dosing variable that acts as the input. Use the “add_dosing_point” method to declare which component is changed at a certain time by a specific amount. You can add as many dosing points as needed.

# Declare dosing algebraic
r1.set_dosing_var(5)

# Add dosing points
r1.add_dosing_point('AH', 100, 0.3)

r1.set_times(0, 600)

r1.simulate()

r1.results.plot('Z')
r1.results.plot('Y')

1.5.7 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, then simply set its absorbing argument to False when declaring the component.
Fig. 1.12: Concentration profile of solution to Tutorial 5

Fig. 1.13: Algebraic state profiles of solution to Tutorial 5
In the plot of the absorbance profile the non-absorbing species then remains zero as you can see in the following results.

```python
from kipet import KipetModel

kipet_model = KipetModel()

r1 = kipet_model.new_reaction('reaction-1')

# Add the model parameters
r1.add_parameter('k1', init=2, bounds=(0.1, 5.0))
r1.add_parameter('k2', init=0.2, bounds=(0.01, 2.0))

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=1e-3)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0, absorbing=False)

# Add the data
r1.add_dataset('D_frame', category='spectral', file='Dij.txt')

# define explicit system of ODEs
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

r1.add_equations(rule_odes)

# Settings
r1.settings.collocation.ncp = 1
r1.settings.collocation.nfe = 60
r1.settings.variance_estimator.use_subset_lambdas = True
r1.settings.variance_estimator.max_iter = 5
r1.settings.variance_estimator.tolerance = 1e-4
r1.settings.parameter_estimator.tee = False

r1.run_opt()
r1.results.show_parameters
r1.results.plot()

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

The estimated parameters are:
k2 0.99999867398201
k1 0.10000000050530382
```

1.5.8 Tutorial 7– Parameter Estimation using concentration data

Files
Fig. 1.14: Concentration profile of solution to Tutorial 6

Fig. 1.15: Absorbance profile of Tutorial 6
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.

```python
from kipet import KipetModel

kipet_model = KipetModel()

r1 = kipet_model.new_reaction('reaction-1')

# Add the model parameters
r1.add_parameter('k1', init=2.0, bounds=(0.0, 5.0))
r1.add_parameter('k2', init=0.2, bounds=(0.0, 2.0))

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=0.001)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0)

# Add data
r1.add_dataset(file='Ex_1_C_data.txt')

# Define the reaction model
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

r1.add_equations(rule_odes)

# Settings
r1.settings.collocation.nfe = 60
r1.settings.parameter_estimator.solver = 'ipopt'

# Run KIPET
r1.run_opt()
r1.results.show_parameters
r1.results.plot()
```

If the component data has been entered into the model before the data, the add_dataset method will automatically check if the component names match the column headers in the dataframe and add them to the model template in the correct category.

```python
# Add data
r1.add_dataset(file='Ex_1_C_data.txt')

# Define the reaction model
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

r1.add_equations(rule_odes)

# Settings
r1.settings.collocation.nfe = 60
r1.settings.parameter_estimator.solver = 'ipopt'

# Run KIPET
r1.run_opt()
r1.results.show_parameters
r1.results.plot()
```

If the user is interested in analyzing the confidence intervals associated with each estimated parameter, the same procedure as shown previously is used. You simply need to use sIPOPT:

```python
r1.settings.parameter_estimator.solver = 'ipopt_sens'
```

This can also 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”.
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.

1.5.9 Tutorial 8 – Variance and parameter estimation with time-dependent inputs using finite element by finite element approach

This tutorial is under construction! Check back shortly.

1.5.10 Tutorial 9 – Interfering species and fixing absorbances

This tutorial is under construction! Check back shortly.

1.5.11 Tutorial 10 – Estimability analysis

Files
Ex_8_estimability.py

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.

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

kipet_model = KipetModel()

r1 = kipet_model.new_reaction('reaction-1')

# Add the model parameters
r1.add_parameter('k1', bounds=(0.1,2))
r1.add_parameter('k2', bounds=(0.0,2))
r1.add_parameter('k3', bounds=(0.0,2))
r1.add_parameter('k4', bounds=(0.0,2))

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=0.3)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0)
r1.add_component('D', state='concentration', init=0.01)
r1.add_component('E', state='concentration', init=0.0)

filename = r1.set_directory('new_estim_problem_conc.csv')
r1.add_dataset('C_frame', category='concentration', file=filename)

# define explicit system of ODEs
def rule_odes(m,t):
    exprs = dict()
    exprs['A'] = -m.P['k1']*m.Z[t,'A']-m.P['k4']*m.Z[t,'A']
    exprs['B'] = m.P['k1']*m.Z[t,'A']-m.P['k2']*m.Z[t,'B']-m.P['k3']*m.Z[t,'B']

    return exprs
```

(continues on next page)
exprs['C'] = m.P['k2']*m.Z[t,'B']-m.P['k4']*m.Z[t,'C']
exprs['D'] = m.P['k4']*m.Z[t,'A']-m.P['k3']*m.Z[t,'D']
exprs['E'] = m.P['k3']*m.Z[t,'B']

return exprs

r1.add_equations(rule_odes)
r1.set_times(0, 20)
r1.create_pyomo_model()

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

e_analyzer = EstimabilityAnalyzer(r1.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':0.5}
sigmas = {'A':1e-10,'B':1e-10,'C':1e-11, 'D':1e-11,'E':1e-11,'device':3e-9}
meas_uncertainty = 0.05

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”

1.5.12 Tutorial 11 – Using the wavelength selection tools

Files Find the file name
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.

```python
from kipet import KipetModel

kipet_model = KipetModel()

r1 = kipet_model.new_reaction('reaction-1')

# Add the model parameters
r1.add_parameter('k1', init=4.0, bounds=(0.0, 5.0))
r1.add_parameter('k2', init=0.5, bounds=(0.0, 1.0))

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=1e-3)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0)

# Add data
r1.add_dataset('D_frame', category='spectral', file='Dij.txt')

# define explicit system of ODEs
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

r1.add_equations(rule_odes)

# Settings
r1.settings.collocation.ncp = 3
r1.settings.collocation.nfe = 60
r1.settings.variance_estimator.use_subset_lambdas = True
r1.settings.variance_estimator.tolerance = 1e-5
r1.settings.parameter_estimator.tee = False

r1.run_opt()
r1.results.show_parameters
r1.results.plot()
```

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

```python
lof = r1.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:

```python
correlations = r1.wavelength_correlation()
```

This function prints a figure that shows the correlations (0, 1) of each wavelength 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:

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:

```
rl.run_lof_analysis()
```

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 wavelengths 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:

<table>
<thead>
<tr>
<th>Wavelengths of less than</th>
<th>Correlation of wavelength</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 correlation removed</td>
<td>1.3759210191412483</td>
</tr>
<tr>
<td>0.2 correlation removed</td>
<td>1.3902630158740596</td>
</tr>
<tr>
<td>0.4 correlation removed</td>
<td>1.4369628529062384</td>
</tr>
<tr>
<td>0.6 correlation removed</td>
<td>1.4585991614309648</td>
</tr>
<tr>
<td>0.8 correlation removed</td>
<td>1.5927062320924816</td>
</tr>
</tbody>
</table>

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:

```
rl.run_lof_analysis(step_size = 0.01, search_range = (0, 0.12))
```

With the additional arguments above, the output is:

<table>
<thead>
<tr>
<th>Wavelengths of less than</th>
<th>Correlation of wavelength</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 correlation removed</td>
<td>1.3759210191412483</td>
</tr>
<tr>
<td>0.01 correlation removed</td>
<td>1.3759210191412483</td>
</tr>
</tbody>
</table>

(continues on next page)
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:

```python
subset = r1.wavelength_subset_selection(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:

```python
subset_results = r1.run_opt_with_subset_lambdas(subset)
# Display the new results
subset_results.show_parameters
# display results
subset_results.plot()
```

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

1.5.13 Tutorial 12 – Parameter estimation over multiple datasets

Files

   Ex_11_multiple_experiments_spectral.py
   Ex_12_multiple_experiments_concentration.py

KIPET allows for the estimation of kinetic parameters with multiple experimental datasets through the MultipleExperimentsEstimator class. This is handled automatically and takes place when the KipetModel instance contains more
Fig. 1.17: Concentration profile for the tutorial example 11

Fig. 1.18: Absorbance profile for the tutorial example 11
than one ReactionModel instance in its models attribute. See the example code below for an overview.

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_12_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:

```python
from kipet import KipetModel

kipet_model = KipetModel()

r1 = kipet_model.new_reaction(name='reaction-1')

# Add the model parameters
r1.add_parameter('k1', init=1.0, bounds=(0.0, 10.0))
r1.add_parameter('k2', init=0.224, bounds=(0.0, 10.0))

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=1e-3)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0)

# define explicit system of ODEs
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

r1.add_equations(rule_odes)

# Add the dataset for the first model
r1.add_dataset(file='Dij_exp1.txt', category='spectral')

# Repeat for the second model - the only difference is the dataset
r2 = kipet_model.new_reaction(name='reaction_2', model_to_clone=r1, items_not_copied='datasets')

# Add the dataset for the second model
r2.add_dataset(file='Dij_exp3_reduced.txt', category='spectral')

kipet_model.settings.general.use_wavelength_subset = True
kipet_model.settings.general.freq_wavelength_subset = 3
kipet_model.settings.collocation.nfe = 100

# If you provide your variances, they need to added directly to run_opt
#user_provided_variances = {'A':1e-10,'B':1e-10,'C':1e-11,'device':1e-6}
"""Using confidence intervals - uncomment the following three lines"""

(continues on next page)
kipet_model.settings.parameter_estimator.solver = 'ipopt_sens'
kipet_model.settings.parameter_estimator.covariance = True
kipet_model.settings.parameter_estimator.scaled_variance = True

"""End of confidence interval section"

kipet_model.run_opt()

for model, results in kipet_model.results.items():
    results.show_parameters
    results.plot()

This outputs the following:
The estimated parameters are:
k2  1.357178
k1  0.279039

Fig. 1.19: Concentration profiles for the tutorial example 12

Fig. 1.20: Absorbance profiles for the tutorial example 12

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.

1.5.14 Tutorial 13 - Using the alternative VarianceEstimator

Files  Ex_13_alternate_method_variances.py
Since the above method that was used in the other problems, described in the initial paper from Chen et al. (2016), can be problematic for certain problems, new variance estimation procedures have been developed and implemented in KIPET version 1.1.01. In these new variance estimation strategies, we solve the maximum likelihood problems directly. The first method, described in the introduction in section 3 involves first solving for the overall variance in the problem and then solving iteratively in order to find how much of that variance is found in the model and how much is found in the device. This tutorial problem can be found in the example directory as “Ex_13_alt_variance_tutorial.py”.

```python
from kipet import KipetModel

kipet_model = KipetModel()

r1 = kipet_model.new_reaction('reaction-1')

# Add the model parameters
r1.add_parameter('k1', init=1.2, bounds=(0.5, 5.0))
r1.add_parameter('k2', init=0.2, bounds=(0.005, 5.0))

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=1e-2)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0)

# define explicit system of ODEs
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

r1.add_equations(rule_odes)

r1.bound_profile(var='S', bounds=(0, 100))

# Add data (after components)
r1.add_dataset(category='spectral', file='varest.csv', remove_negatives=True)

# Settings
r1.settings.general.no_user_scaling = True
r1.settings.variance_estimator.tolerance = 1e-10
r1.settings.parameter_estimator.tee = False
r1.settings.parameter_estimator.solver = 'ipopt_sens'

After setting the problem up in the normal way, we then call the variance estimation routine with a number of new options that help to inform this new technique.

```python
r1.settings.variance_estimator.method = 'alternate'
r1.settings.variance_estimator.secant_point = 5e-4
r1.settings.variance_estimator.initial_sigmas = 5e-5
```
Included in this tutorial problem is the ability to compare solutions with the standard Chen approach as well as to compare the solutions to the generated data. One can see that both approaches do give differing solutions. And that, in this case, the new variance estimator gives superior solutions.

1.5.15 Tutorial 14 – Unwanted Contributions in Spectroscopic data

Files
Ex_15_time_variant_unwanted_contributions.py
Ex_15_time_invariant_unwanted_contributions.py
Ex_15_multiple_experiments_unwanted_contributions.py

In many cases, there may be unwanted contributions in the measured spectra, which may come from instrumental variations (such as the baseline shift or distortion) or from the presence of inert absorbing interferences with no kinetic behavior. Based on the paper of Chen, et al. (2019), we added a new function to KIPET in order to deal with these unwanted contributions.

The unwanted contributions can be divided into the time invariant and the time variant instrumental variations. The time invariant unwanted contributions include baseline shift, distortion and presence of inert absorbing interferences without kinetic behavior. The main time variant unwanted contributions come from data drifting in the spectroscopic data. Beer-Lambert’s law can be modified as,

\[ D = CS^T + G + E \]

where \( G \) is the unwanted contribution term.

The time invariant unwanted contributions can be uniformly described as the addition of a rank-1 matrix \( G \), given by the outer product,

\[ G = e g^T, e = [1 \ 1 \ \cdots \ 1]^T_{(npxl)}, g = [g_1 \ g_2 \ \cdots \ g_{nvw}]^T, G \in \mathbb{R}^{npxl \times nvw} \]

where the vector \( g \) represents the unwanted contribution at each sampling time. According to Chen’s paper, the choices of objective function to deal with time invariant unwanted contributions depends on the rank of kernel of \( \Omega_{sub} \) matrix (rko), which is composed of stoichiometric coefficient matrix \( S_t \) and dosing concentration matrix \( Z_{in} \). (detailed derivation is omitted.) If rko > 0, \( G \) can be decomposed as,

\[ G = CS_g^T, \quad S_g = [s_1^g, s_2^g, \ldots, s_{nvw}^g] \in \mathbb{R}^{nc \times nvw} \]

Then the Beer-Lambert’s law can be rewritten as,

Thus, the original objective function of the parameter estimation problem doesn’t need to change while the estimated absorbance matrix would be \( S+S_g \) and additional information is needed to separate \( S \) and \( S_g \).

If rko = 0, \( G \) cannot to decomposed. Therefore, the objective function of the parameter estimation problem should be modified as,
For time variant unwanted contributions, $G$ can be expressed as a rank-1 matrix as well, and the objective of problem is modified as follows,

$$G = qg^T, \quad q = \begin{bmatrix} q_1 & q_2 & \ldots & q_{ntp} \end{bmatrix}^T, \quad g = \begin{bmatrix} g_1 & g_2 & \ldots & g_{nwv} \end{bmatrix}^T$$

and the objective of problem is modified as follows,

where the time variant unwanted contributions are considered as a function of time and wavelength. In addition, since there are no constraints except bounds to restrict $q(t)$ and $g(t)$, this will lead to nonunique values of these two variables and convergence difficulty in solving optimization problem. Therefore, we force $q(t_{ntp})$ to be 1.0 under the assumption that $q(t_{ntp})$ is not equal to zero to resolve the convergence problem.

Users who want to deal with unwanted contributions can follow the following algorithm based on how they know about the unwanted contributions. If they know the type of the unwanted contributions is time variant, assign `time_variant_G = True`. On the other hand, if the type of the unwanted contributions is time invariant, users should set `time_invariant_G = True` and provide the information of $St$ and/or $Z_in$ to check $rko$. However, if the user have no idea about what type of unwanted contributions is, assign `unwanted_G = True` and then KIPET will assume it’s time variant.

Please see the following examples for detailed implementation. The model for these examples is the same as “Ex_2_estimation.py” with initial concentration: $A = 0.01$, $B = 0.0$, and $C = 0.0$ mol/L.

The first example, “Ex_15_time_invariant_unwanted_contribution.py” shows how to estimate the parameters with “time invariant” unwanted contributions. Assuming the users know the time invariant unwanted contributions are involved, information of $St$ and/or $Z_in$ should be inputed as follows,

```python
St = dict()
St["r1"] = [-1,1,0]
St["r2"] = [0,-1,0]
```

# In this case, there is no dosing time.
# Therefore, the following expression is just an input example.

```python
Z_in = dict()
Z_in["t=5"] = [0,0,5]
```

Next, add the option `G_contribution` equal to “time_invariant_G = True” and transmit the $St$ and $Z_in$ (if users have $Z_in$ in their model) matrix when calling the “run_opt” method to solve the optimization problem.

```python
r1.settings.parameter_estimator.G_contribution = 'time_invariant_G'
r1.settings.parameter_estimator.St = St
r1.settings.parameter_estimator.Z_in = Z_in
```
\[
\min \sum_{i=1}^{nlp} \sum_{l=1}^{mv} \left( d_{i,l} - \sum_{k=1}^{nc} c_k(t_i) s_k(\lambda_l) - q_r(t_i) g(\lambda_l) \right)^2
\]
The next example, “Ex_15_time_variant_unwanted_contribution.py” shows how to solve the parameter estimation problem with “time variant” unwanted contribution in the spectra data. Simply add the option G_contribution equal to “time_variant_G” to the arguments before solving the parameter estimation problem.

```python
r1.settings.parameter_estimator.G_contribution = 'time_variant_G'
```

As mentioned before, if users don’t know what type of unwanted contributions is, set G_contribution equal to ‘time_variant’.

In the next example, “Ex_15_estimation_mult_exp_unwanted_G.py”, we also show how to solve the parameter estimation problem for multiple experiments with different unwanted contributions. The methods for building the dynamic model and estimating variances for each dataset are the same as mentioned before. In this case, Exp1 has “time invariant” unwanted contributions and Exp2 has “time variant” unwanted contributions while Exp3 doesn’t include any unwanted contributions. Therefore, we only need to provide unwanted contribution information for each Reaction-Model separately as you would for individual models.

Users may also wish to solve the estimation problem with scaled variances. For example, if the estimated variances are (“A”: 1e-8, “B”: 2e-8, “device”: 4e-8) with the objective function,

\[
\text{Min} \quad \frac{(D - C_A S_A - C_B S_b)^2}{4e-8} + \frac{(C_A - Z_A)^2}{1e-8} + \frac{(C_B - Z_B)^2}{2e-8}
\]

this option will scale the variances with the maximum variance (i.e. 4e-8 in this case) and thus the scaled variances become (“A”: 0.25, “B”: 0.5, “device”: 1.0) with modified objective function,

\[
\text{Min} \quad \frac{(D - C_A S_A - C_B S_b)^2}{1.0} + \frac{(C_A - Z_A)^2}{0.25} + \frac{(C_B - Z_B)^2}{0.5}
\]

This scaled_variance option is not necessary but it helps solve the estimation problem for multiple datasets. It’s worth trying when ipopt gets stuck at certain iteration.

```python
kipet_model.settings.general.scale_variances = True
```

**1.5.16 Tutorial 15 – Simultaneous Parameter Selection and Estimation**

**Files** Ex_16_reduced_hessian_parameter_selection.py

The complex models used in reaction kinetic models require accurate parameter estimates. However, it may be difficult to make accurate estimates for all of the parameters. To this end, various techniques have been developed to identify parameter subsets that can best be estimated while the remaining parameters are fixed to some initial value. The selection of this subset is still a challenge.

One such method for parameter subset selection was recently developed by Chen and Biegler (2020). This method uses a reduced hessian approach to select parameters and estimate their values simultaneously using a collocation approach. Parameter estimability is based on the ratio of their standard deviation to estimated value, and a Gauss-Jordan elimination method strategy is used to rank parameter estimability. This has been shown to be less computationally demanding than previous methods based on eigenvalues. For more details about how the algorithm works, the user is recommended to read the article “Reduced Hessian Based Parameter Selection and Estimation with Simultaneous Collocation Approach” by Weifeng Chen and Lorenz T. Biegler, AIChE 2020.

In Kipet, this method is implemented using the EstimationPotential module. It is currently separate from the EstimabilityAnalyzer module used otherwise for estimability (see Tutorial 12). Kipet can now handle complementary state data, such as temperature and pressure, in its analysis. This should improve the user experience and lead to more robust results.
This module is used in a slightly different manner than other modules in Kipet. The EstimationPotential class requires the TemplateBuilder instance of the model as the first argument (the models are declared internally). This is followed by the experimental data. Yes, this form of estimability analysis requires experimental data because the analysis depends on the outputs. For illustration purposes, the example CSTR problem in this example includes simulated data at the "true" parameter values. Optional arguments include simulation_data, which takes a Results instance as input. This is recommended for complex systems that require good initializations. If no simulation data is provided, the user can use the argument simulate_start to select whether a simulation should be performed internally; performance may vary here, so it is usually better to provide your own simulated data as above.

This tutorial has two examples based on the CSTR example from the paper by Chen and Biegler (2020).

The code for the entire problem is below:

```python
from pyomo.environ import exp
from kipet import KipetModel

kipet_model = KipetModel()

r1 = kipet_model.new_reaction('cstr')

# Perturb the initial parameter values by some factor
factor = 1.2

# Add the model parameters
r1.add_parameter('Tf', init=293.15*factor, bounds=(250, 400))
r1.add_parameter('Cfa', init=2500*factor, bounds=(0, 5000))
r1.add_parameter('rho', init=1025*factor, bounds=(800, 2000))
r1.add_parameter('delH', init=160*factor, bounds=(0, 400))
r1.add_parameter('ER', init=255*factor, bounds=(0, 500))
r1.add_parameter('k', init=2.5*factor, bounds=(0, 10))
r1.add_parameter('Tfc', init=283.15*factor, bounds=(250, 400))
r1.add_parameter('rhoc', init=1000*factor, bounds=(0, 2000))
r1.add_parameter('h', init=3600*factor, bounds=(0, 5000))

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=1000, variance=0.001)
r1.add_component('T', state='state', init=293.15, variance=0.0625)
r1.add_component('Tc', state='state', init=293.15, variance=1)

r1.add_dataset(file='cstr_t_and_c.csv')

constants = {
    'F' : 0.1, # m^3/h
    'Fc' : 0.15, # m^3/h
    'Ca0' : 1000, # mol/m^3
    'V' : 0.2, # m^3
    'Vc' : 0.055, # m^3
    'A' : 4.5, # m^2
    'Cpc' : 1.2, # kJ/kg/K
    'Cp' : 1.55, # kJ/kg/K
}

# Make it easier to use the constants in the ODEs
C = constants

# Define the model ODEs
def rule_odes(m,t):
    (continues on next page)
Ra = m.P['k']*exp(-m.P['ER']/m.X[t,'T'])*m.Z[t,'A']
exprs = dict()
exprs['A'] = C['F']/C['V']*(m.P['Cfa']-m.Z[t,'A']) - Ra
exprs['T'] = C['F']/C['V']*(m.P['Tf']-m.X[t,'T']) + m.P['delH']/(m.P['rho'])/C['Cp']*Ra - m.P['h']*(m.P['rho'])/C['Cp']/(m.P['rho'])/C['Cp']*(m.X[t,'T'] - m.X[t,'Tc'])
exprs['Tc'] = C['Fc']/C['Vc']*(m.P['Tfc']-m.X[t,'Tc']) + m.P['h']*(m.P['rho'])/C['Cpc']/(m.P['rho'])/C['Vc']*(m.X[t,'T'] - m.X[t,'Tc'])
return exprs

r1.add_equations(rule_odes)

To start the simultaneous parameter selection and estimation routine, simply use the estimate method.

```python
# Run the model reduction method
results = r1.reduce_model()

# results is a standard ResultsObject
results.plot(show_plot=with_plots)
```

This example is “Ex_16_reduced_hessian_parameter_selection.py” and can be found with the other examples.

The data comes from simulation and is performed with 50 finite elements and 3 collocation points. Thus, there are 151 potential times for a measurement. Three points are chosen and the concentration data (Z_data) is limited to these three points. The complete experimental data (50 temperature points and three concentration measurements) are concatenated together. Kipet can handle the discrepancies in measurement times.

![Concentration Profile](image)

Fig. 1.21: Concentration profiles for the tutorial example 16. Notice the addition of the three “experimental” points.

1.5.17 Tutorial 16 – Custom Data and Objective Functions

Files Ex_17_custom_objective.py

In the case where you have data that does not fit into one of the predefined categories, such as ratios between components for examples, you can still use this data to fit kinetic models in KIPET. How to do this is shown in the following code. The data needs to be entered into the model with the category ‘custom’ and labeled with a name, in this case ‘y’. You then need to declare an algebraic variable of the same name and define the term to be used in fitting this data. The algebraic relationship will then be added to the objective function using a least squares term. The only extra
The requirement is to tell the model that this algebraic variable ‘y’ represents this new objective term. This is done using the method ‘add_objective_from_algebraic’ with ‘y’ as the sole argument.

In this example, concentration data for component A is available as normal. However, only the fraction of B compared to C (B/(B + C)) is provided for these components. This data can also be used in fitting the model.

```python
import kipet

kipet_model = kipet.KipetModel()
full_data = kipet.read_file(kipet.set_directory('ratios.csv'))
r1 = kipet_model.new_reaction('reaction-1')

# Add the model parameters
r1.add_parameter('k1', init=5.0, bounds=(0.0, 10.0))
r1.add_parameter('k2', init=5.0, bounds=(0.0, 10.0))

# Declare the components and give the initial values
r1.add_component('A', state='concentration', init=1.0)
r1.add_component('B', state='concentration', init=0.0)
r1.add_component('C', state='concentration', init=0.0)

r1.add_dataset(data=full_data[['A']], remove_negatives=True)
r1.add_dataset('y_data', category='custom', data=full_data[['y']])

# Define the reaction model
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

r1.add_equations(rule_odes)

# To use custom objective terms for special data, define the variable as an algegraic and provide the relationship between components
```

(continues on next page)
r1.add_algebraic_variables('y', init = 0.0, bounds = (0.0, 1.0))

def rule_algebraics(m, t):
    r = list()
    r.append(m.Y[t, 'y']*(m.Z[t, 'B'] + m.Z[t, 'C']) - m.Z[t, 'B'])
    return r

r1.add_algebraics(rule_algebraics)

# Add the custom objective variable to the model using the following method:
r1.add_objective_from_algebraic('y')

r1.run_opt()
r1.results.show_parameters
r1.results.plot()

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.

1.6 Additional Functions

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 other subtleties that may be useful to the user in implementing and improving their models.

1.6.1 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.

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:

```python
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:

```python
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:

```python
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:

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

or if you have a csv file:

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

**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:

```python
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.

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)

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.

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.
**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:

```python
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.

**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:

```python
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.

**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):

```python
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.

**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:

```python
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.
1.6.2 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:

```python
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
```

1.6.3 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:

```python
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
```

1.6.4 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:

```python
def run_opt(self, solver, **kwds):
    (continues on next page)
```
"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.
- 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.

1.6.5 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 $D_{\text{bar}}(i,j)$, 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

1.6.6 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:

```python
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:

```python
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:

```python
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:
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:

```python
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.

```python
builder.addParameter('k1', init=1.0, bounds=(0.0, 10.0))
```

An example can be found in Ex_2_estimation_factoryTempV.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

```python
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. An 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:

```python
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:

```python
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:

```python
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.

In some cases it can be useful to provide expected optimal parameter values and ensure that the estimated parameters stay close to these values. For that purpose, it is possible to add optional quadratic penalty terms to the objective and define the expected parameter values and corresponding penalty weights, e.g.
where in `ppenalty_dict` you define the expected optimal values and in `ppenalty_weights` you define the corresponding weights. These dictionaries should then be handed to the ParameterEstimator setting the `penaltyparam` option to True as well, i.e.

```python
results_pyomo = p_estimator.run_opt('ipopt',
   tee=True,
   solver_opts=options,
   variances=sigmas,
   with_d_vars=True,
   penaltyparam=True,
   ppenalty_dict=ppenalty_dict,
   ppenalty_weights=ppenalty_weights)
```

In case one wants to check the eigenvalues of the reduced Hessian to check whether the estimates have large variances, set the option `eigredhess2file` option to True, i.e.

```python
eigredhess2file=True
```

handing it to the ParameterEstimator. Note that to use this option you have to solve the problem with sensitivities, i.e. the solver ‘ipopt_sens’ or ‘k_aug’ has to be called.

If you have additional HPLC or UPLC data available, you can make use of that data for the parameter estimation as well. The H/UPLC data needs to be added to the model in a similar fashion as the concentration data. However, the measurements are area percentages and should add up to 1 at each time point. Furthermore, the species that are absorbed by H/UPLC data need to be defined in a set called `huplcabs`, where they should have the same names as the columns in the data file. You then hand those to the function

```python
filename2 = os.path.join(dataDirectory, 'UPLCdata2.csv')
add_frame = read_concentration_data_from_csv(filename2)

builder.add_huplc_data(add_frame)
opt_model = builder.create_pyomo_model(0.0,10.0)
huplcabs = ['A','C']
builder.set_huplc_absorbing_species(opt_model, huplcabs)
```

You should specify a device variance for the H/UPLC data as well. If you do not declare one by default it will be set to 1.

```python
# Here we assume that the UPLC data has the same variance as the IR data:
sigmas['device-huplc'] = sigmas['device']
```

The lack of fit related to the UPLC data can be calculated using

```python
lof2 = p_estimator.lack_of_fit_huplc()
```

An example is provided in `Ad_2_estimation_uplc.py`.

In case one wants to determine the initial values of complementary state variables as degrees of freedom of the parameter estimation problem as well, after creating the model, the following additions have to be made to the Template-Builder called `builder` here:
modelexpl = builder.create_pyomo_model(start_time1, end_time1)
initextra = {'A'}  # to define to be estimated initial values for extra states
builder.add_init_extra('A', init=1e-3, bounds=(0.0, 0.1))
builder.set_estinit_extra_species(modelexpl, initextra)  # to define to be estimated initial values for extra states

If you want to solve a parameter estimation problem for a single experiment, you will have to create the model again after this and hand it to the ParameterEstimator. When solving a parameter estimation problem for multiple experiments, you can hand the builder as an argument to the run_parameter_estimation function after calling the MultipleExperimentsEstimator on the datasets.

pest = MultipleExperimentsEstimator(datasets)
results_pest = pest.run_parameter_estimation(solver = 'k_aug',
                                          tee=True,
                                          nfe=nfe,
                                          ncp=ncp,
                                          covariance = True,
                                          solver_opts = options,
                                          start_time=start_time,
                                          end_time=end_time,
                                          spectra_problem = False,
                                          sigma_sq=variances,
                                          builder = builder_dict)

An example with some initialization options commented out is provided in Ad_13_estimation_mult_exp_cone_conf_initest.py. In case of initialization with previous results, the upper additions always have to be made before calling the ParameterEstimator and not before running a simulation or variance estimation. The builders for these two are different in this case. You can find more explanation in the example.

1.7 References


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


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


CHAPTER 2

Indices and tables

• genindex
• modindex
• search
KIPET development is hosted on GitHub and we welcome feedback and questions there:
https://github.com/salvadorgarciamunoz/KIPET
KIPET makes use of Pyomo as the algebraic modelling language and much of the syntax can be found here: