KIPET Documentation

Release 0.2.4

mchlshort

Jul 05, 2021

CONTENTS

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

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation Guide

KIPET is written in the Python programming language and requires at least version 3.8 for best performance. Thus, to use KIPET, Python needs to be installed on your workstation.

Note: Support for Python 2.7 has been dropped in the most recent versions of KIPET!

This documentation does not include a detailed decription of how to install Python. There are enough sources on the internet where you can find detailed instructions on how to install it, such as this detailed guide from Real Python that covers all the major operating systems.

1.1.1 Installation

The latest versions of KIPET are conveniently provided as Python packages which are hosted in the usual locations. You can use any resource you wish to manage the installation, including virtualenv, poetry, conda, etc.

PyPi Package

KIPET can be installed with the standard packaging manager, pip:

```
pip install kipet
```

If you prefer to use poetry to manage your packages, you can also use

```
poetry add kipet
```

Anaconda Package

If you prefer to use Anaconda, KIPET can be installed using:

```
conda install -c kwmcbride kipet
```

Note: The anaconda packages have the benefit of already including pynumero, which is needed (but not required) for some of the methods included in KIPET. You will need to compile these on your own if you choose to install KIPET

using a different method. See the pynumero readme for more information. Otherwise, you can also use k_aug for these methods as well.

GitHub

Additionally, KIPET may be installed directly from the repository (for example, if using poetry, simply install the desired branch (#branch) in the following manner):

```
poetry add git+https://github.com/salvadorgarciamunoz/kipet#master
```

Naturally you can simply clone or download the repository if you wish. If you desire to further develop KIPET for your own needs, this method is recommended.

```
cd <installation directory>
git clone https://github.com/salvadorgarciamunoz/kipet.git
cd kipet
python setup.py install
```

If you would like to contribute to KIPET, this is the recommended installation route.

1.1.2 Installing IPOPT

To use KIPET for parameter fitting, you need to have a solver installed that can solver NLPs. 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.

The installation methods only show how to install Ipopt. There are several third party linear solvers that Ipopt requires and these also need to be installed. See the COIN-OR link above for more information.

Linux/MacOS Installation

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

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

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

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

Then go into the directory we just created:

```
cd CoinIpopt
```

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

```
mkdir build
```

and enter this directory:

```
cd build
```

Then we run the configure script:

```
../configure
```

make the code

```
make
```

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

```
make test
```

Finally we install IPOPT:

```
make install
```

Microsoft Windows

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

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

Run an example (explained in the next section) to test if it works. This method should also include a functioning version of sIpopt and so the next step is not necessary unless another method of installation is used. If trouble is experienced using this approach other methods can be used and they are detailed in the Introduction to IPOPT document.

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

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

Note: 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.

1.1.3 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 installation guide can be found within the same folder as this documentation on the Github page or on David M. Thierry's Github page.

Note: If you are using a Linux OS, you can try a script that automatically installs k_aug for you.

1.1.4 Examples and Tutorials

All of the example problems can be easily downloaded from the examples repository:

```
cd <example directory>
git clone https://github.com/kwmcbride/kipet_examples.git
```

To validate your installation, you can now run the test script included with the examples:

```
cd <example directory>/kipet_examples
python run_examples.py
```

1.1.5 Validation of the Package

If you want to validate your installation of KIPET, you can download the example problems found in the examples repository. You can then run all examples with the following:

```
python kipet_examples/run_examples.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, you can simply ignore these failures.

1.1.6 Updating KIPET

New versions of KIPET can be updated using the respective package manager you used to install KIPET.

1.1.7 Troubeshooting

Installation via Anaconda Fails

Some users may not be able to install KIPET using Anaconda. The issue being raised is usually in regard to the version of Python not being correct. This is being worked on, but you can still install KIPET using the Ananconda Navigator GUI or by simply installing using pip within your conda environment.

Windows PATH Management

If there are issues found with running examples it may be necessary in Windows to add Python to the PATH environment 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:UsersUsernameAnaconda3 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!

Issues

If you encounter any issues, please report them on our Github page.

1.2 Overview

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 documentatation 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.

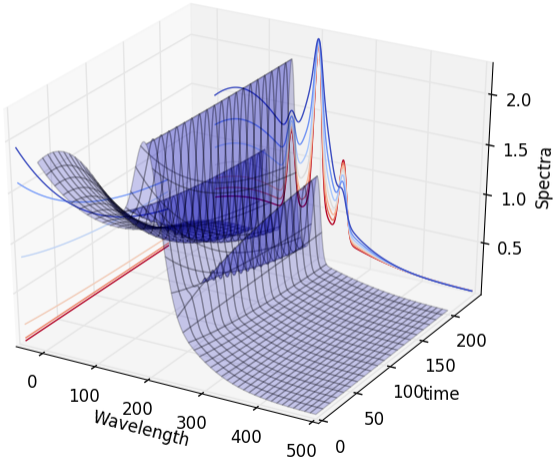


Fig. 1.1: Visualization of a multi-wavelength spectroscopic dataset

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
- 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.)

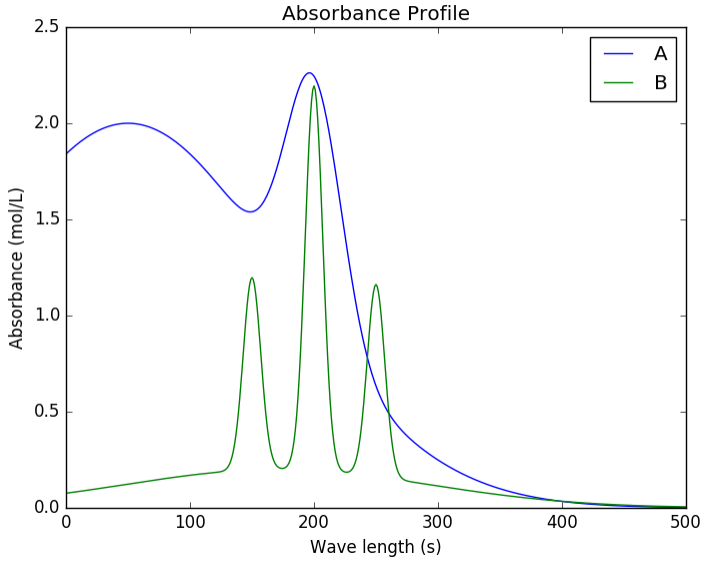


Fig. 1.2: Example of the decomposition of the spectra in single component absorbance and concentration profiles (absorbance)

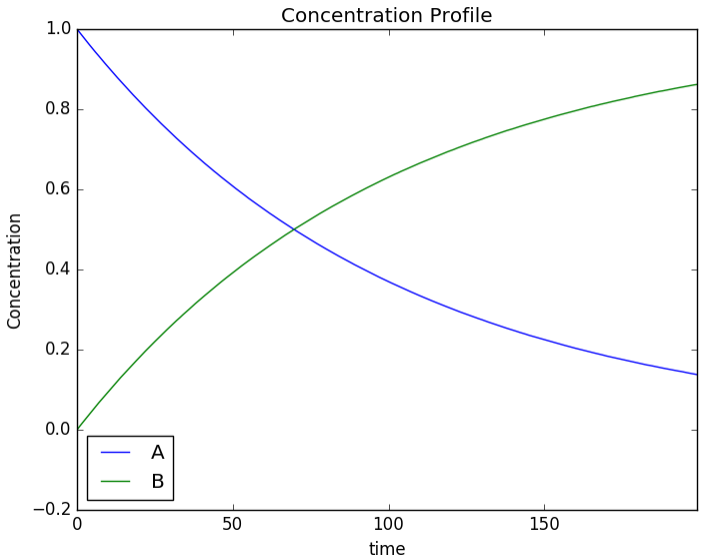


Fig. 1.3: Example of the decomposition of the spectra in single component absorbance and concentration profiles (concentration)

- Visualize results

In the sections that follow, this document provides guidelines on how to make use of KIPET. A detailed installation guide, a getting started section, and many examples are provided.

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 Garcia-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).

1.2.1 Citing KIPET

C. Schenk, M. Short, J.S. Rodriguez, D. Thierry, L.T. Biegler, S. García-Muñoz, W. Chen (2020) Introducing KIPET: A novel open-source software package for kinetic parameter estimation from experimental datasets including spectra, Computers & Chemical Engineering, 134, 106716. https://doi.org/10.1016/j.compchemeng.2019.106716

M. Short, L.T. Biegler, S. García-Muñoz, W. Chen (2020) Estimating variances and kinetic parameters from spectra across multiple datasets using KIPET, Chemometrics and Intelligent Laboratory Systems, https://doi.org/10.1016/j.chemolab.2020.104012

M. Short, C. Schenk, D. Thierry, J.S. Rodriguez, L.T. Biegler, S. García-Muñoz (2019) KIPET–An Open-Source Kinetic Parameter Estimation Toolkit, Computer Aided Chemical Engineering, 47, 299-304.

1.3 Background

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

1.3.1 General modeling strategy and method

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

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

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.

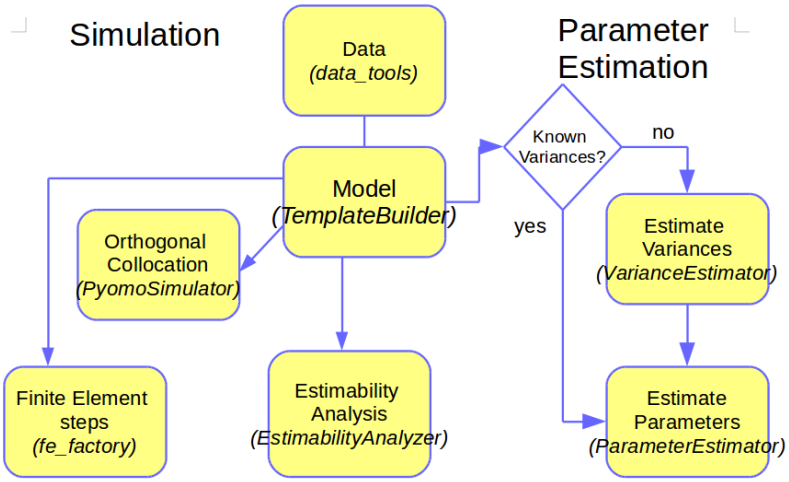


Fig. 1.4: The steps/modules involved in setting up a KIPET model

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.

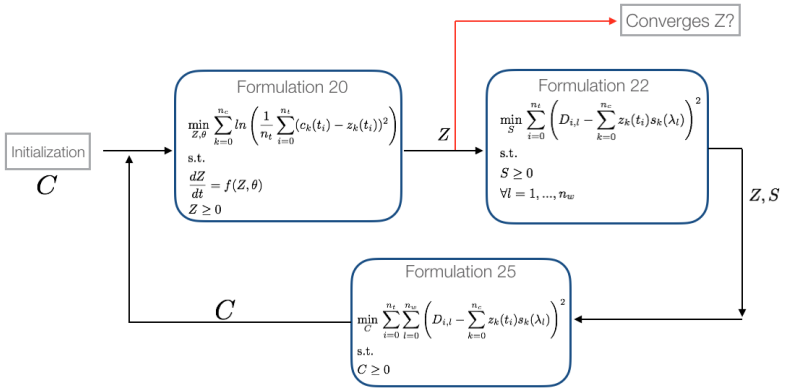


Fig. 1.5: The VarianceEstimator class algorithm 1 from Chen et al. (2016)

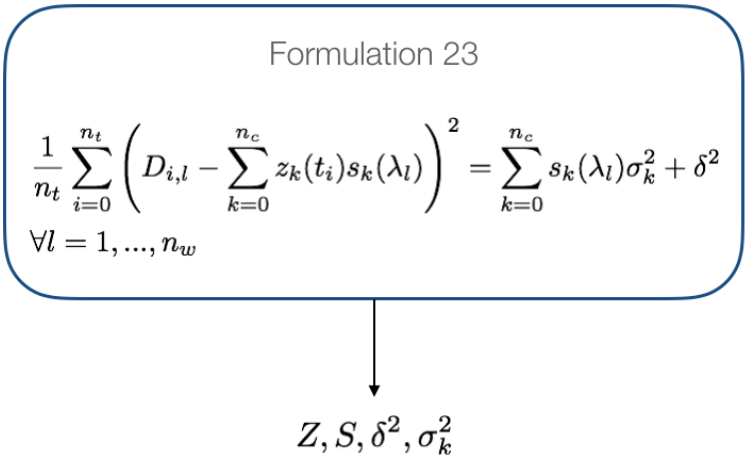


Fig. 1.6: Variance estimation equations

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

$$\min \log \left(\sum_i \tilde{\epsilon}_i^\top \tilde{\epsilon}_i \right)$$

where

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

Then we set:

$$v^2 = \sum_i \tilde{\epsilon}_i^\top \tilde{\epsilon}_i / (n_{wp} \cdot n_{tp})$$

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

$$v^2 = \sum_{l=1}^{n_{wp}} \sum_{k=1}^{n_c} \left(\delta^2 + \sigma^2 s_{kl} \right) / n_{wp}$$

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

$$\min \ n_{wp} \log \left(\sum_i \epsilon_i^\top \epsilon_i \right) + \sum_i \frac{\eta_i^\top \eta}{2\sigma^2}$$

and then we are able to determine delta from:

$$\delta^2 = \sum_i \epsilon_i^\top \epsilon_i / (n_{wp} \cdot n_{tp})$$

Following this we can evaluate:

$$f(\sigma_p) = v^2 - \delta^2 - \sigma_p^2 \left(\sum_{l=1}^{n_{wp}} \sum_{k=1}^{n_c} s_{kl} / n_{wp} \right)$$

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 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:

$$\min \frac{1}{2} n_{tp} \cdot n_{wp} \log \left(\sum_{i=1}^{n_{tp}} \sum_{l=1}^{n_{wp}} \left(d_{il} - \sum_{k=1}^{n_c} z_k(t_i) s_k(\lambda_l) \right)^2 / (n_{tp} \cdot n_{wp}) \right)$$

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:

$$\min \frac{1}{2\delta^2} \sum_{i=1}^{n_{tp}} \sum_{l=1}^{n_{wp}} \left(d_{il} - \sum_{k=1}^{n_c} z_k(t_i) s_k(\lambda_l) \right)^2 \tag{1.1}$$

$$+ \frac{n_{tp} n_c}{2} \log \left(\sum_{i=1}^{n_{tp}} \sum_{k=1}^{n_c} \left(c_k(t_i) - z_k(t_i) \right)^2 / (n_{tp} n_c) \right) \tag{1.2}$$

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:

$$\min \sum_{i=1}^{n_{tp}} \sum_{l=1}^{n_{wp}} \left(d_{il} - \sum_{k=1}^{n_c} c_k(t_i) s_k(\lambda_l) \right)^2 / \delta^2 \tag{1.3}$$

$$+ \sum_{i=1}^{n_{tp}} \sum_{k=1}^{n_c} \left(c_k(t_i) - z_k(t_i) \right)^2 / \sigma_k^2 \tag{1.4}$$

$$\text{s.t.} \sum_{m=0}^{k} \dot{l}_m(\tau) z_{jm} - h_j \cdot \mathbf{f}(z_{jm}, \theta) = 0, \quad j = 1..n_{fe}, \; m = 1..K \tag{1.5}$$

$$z^K(t_i) = \sum_{m=0}^{K} l_m(\tau) z_{jm}, \quad \tau = (t_i - tp_{j-1}) / (tp_j - tp_{j-1}) \tag{1.6}$$

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.

1.4 Getting Started with KIPET

1.4.1 Creating a Model Instance

KIPET is comprised of many modules working in the background. Only two object classes are needed on the user's end. These are the ReactionModel and the ReactionLab classes.

All reactions are modeled as ReactionModel objects. If you are only considering a single reaction system or experiment, the simplest way to use KIPET is through a single ReactionModel instance. To do this, simply import the ReactionModel class from kipet and create an instance:

```
from kipet import ReactionModel
r1 = ReactionModel('model-1')
```

1.4.2 Model Components

The ReactionModel class contains all of the methods necessary to use KIPET for a single model with a single dataset. You 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 **parameter** method. For example, if we have a three component system with two reactions in series:

$$\mathrm{A} \xrightarrow{r_A} \mathrm{B} \xrightarrow{r_B} \mathrm{C}$$

$$\mathrm{r}_A = k_1 C_A$$
$$\mathrm{r}_B = k_2 C_B$$

This system has two reactions and if we model them as simple first order reactions at constant temperature, we only have two parameters to fit: the reaction rate constants k1 and k2. This is done using the **parameter** method of the ReactionModel instance.

```
k1 = r1.parameter('k1', value=2)
k2 = r1.parameter('k2', value=0.2)
```

Thus, two parameter are added to **r1**: **k1** and **k2** with initial values 2 and 0.2, respectively. If you perform a simulation, these values will be the fixed parameter values in these models. For a full reference of parameter options, see parameter.

Since our system has three components, A, B, and C, these need to be declared as well. In KIPET components are understood as being those species (chemicals or biological components) that are measured in concentrations. Components of this nature are added to the ReactionModel using the **component** method. More specifically, these are the components that can be measured using spectroscopic means. Under the hood, each component is treated as a state variable. Each component requires at least a name and an initial value. In our current example, the components can be defined as the following:

```
A = r1.component('A', value=1)
B = r1.component('B', value=0.0)
C = r1.component('C', value=0.0)
```

Here you can see that only species A is present at the start of the reaction and the other components B and C are not.

If you were to add complementary states such as temperature, pressure, etc., this can be done using the **state** method. This should be used for all states that are not measured using concentration. For example, say there is a change in temperature during the reaction. In order to model this temperature change, you would need to add a temperature state variable to the model:

```
T = r1.state('T', value=500, units='K')
```

Notice here that you can optionally add the units for any model component using the keyword argument units. KIPET has tools that ensure that units are converted to proper values and also checks for inconsistant unit types.

Note: Every ReactionModel will have a volume state created automatically. This is to enusre that some features of KIPET that depend on volume always work properly. Thus, the variable name V is reserved for the volume state and an error will be raised if you try to name a model component with this name. If you need to modify the volume state (such as defining initial values and units), use the **volume** method.

```
r1.volume(value=0.45, units='mL')
```

In the same manner, model constants can be generated. If you are using dimensionless units or do not care to check the units, you can simply add constants into the expressions (next section) using their numerical values. Suppose you have a constant feed to the reactor of species A of 2 moles per liter per minute. This could be added to the model using the following:

```
C_Ain = r1.constant('C_Ain', value=2, units='M/min')
```

Another component that can be declared for use in a ReactionModel is a step function. The **step** method is a convenience way to add on/off decisions or steps during the reaction time. For example, if the constant feed rate of A given above is only to last for 5 minutes before being turned off, this behavior can be captured using a step component:

```
A_step = r1.step('A_step', time=5, fixed=True, switch='off')
```

Here you can see the intuitive nature of adding step variables. The first argument is the name, followed by the time where the step occurs, whether the time is fixed at 5 or not (if False, it means that the step time is variable to be fit in the parameter estimation), and the direction of the step (in this case we turn the A feed off). The next step is simply to combine the flow rate of A with its step function during expression building.

```
# The flowrate of A into the reactor is
C_Ain * A_step
```

Step functions can be chained together for more complex on and off behaviors if needed. For example, if you needed to model that the flowrate of A into the reactor starts again at 10 minutes, this can be done in a similar manner as before:

```
A_step_2 = r1.step('A_step', time=10, fixed=True, switch='on')
```

And then the steps can be combined into a single variable (actually an expression now):

```
A_step_all = A_step + A_step_2
```

Note: An improved version for chaining step functions is being worked on to simplify this procedure.

1.4.3 Expressions

For dynamic systems like chemical reactions, we necessarily work with ODEs. Each component and states is automatically assigned an accompanying ODE in KIPET with a default value of zero. If you forget to assign a more specific ODE, it will simply remain constant and not result in an error.

KIPET handles two types of expressions: ODEs and Algebraics. The key difference lies in how each is modeled in KIPET. In our current example, there are two reactions (A –> B and B –> C). These can be constructed as:

The next step is to provide the reaction kinetics. The five variables that were defined above (k1, k2, A, B, C) are all place holder Pyomo variables that can be used to construct expressions. This makes building expressions very simple in KIPET. Expressions can be used in either ODEs or algebraic expressions. There are several ways to add such expressions to KIPET.

```
rA = k1 * A
rB = k2 * B
```

or as

```
rA = r1.add_reaction('rA', k1*A)
rB = r1.add_reaction('rB', k2*B)
```

or equivalently

```
rA = r1.add_expression('rA', k1*A, is_reaction=True)
rB = r1.add_expression('rB', k2*B, is_reaction=True)
```

where the **add_reaction** method simply wraps the **add_expression** method and sets is_reaction to True for you. This syntax is simpler to use in creating the model. Once the reactions and other possible expressions have been generated, the ODEs can be created and added to the ReactionModel:

```
r1.add_ode('A', -rA)
r1.add_ode('B', rA - rB)
r1.add_ode('C', rB)
```

If you would prefer to use a stoichiometric matrix to build the system of equations for the reactions, this is possible as well:

```
rA = r1.add_reaction('rA', k1*A, description='Reaction A' )
rB = r1.add_reaction('rB', k2*B, description='Reaction B' )

stoich_data = {'rA': [-1, 1, 0],
               'rB': [0, -1, 1]}

r1.reactions_from_stoich(stoich_data, add_odes=True)
```

Note the form of the stoichiometric matrix. It takes the reaction name as the key and a list of stoichiometric coefficients as the the values. If you provide the keys as the components instead, KIPET will automatically detect this and still build the appropriate reaction network. The add_odes keyword argument is passed as True if the ODEs are based solely on the reaction kinetics. If you with to add additional terms to the ODEs (such as to account for volume changes), you need to set add_odes to False and use the returned dictionary of reaction ODEs to augment the expressions (see *Simulation of Advanced Reaction System with Algebraic equations (II)*).

For example, say we are feeding C to a reactor and need to take this into account after we have constructed the system of reactions using the stoiciometric matrix. Simply set add_odes to False and use the returned dictionary of ODEs (here RE) and simply add the volumetric change to the existing ODE. After you do this, the ODEs still need to be added to the ReactionModel which can be done using the **add_odes** method.

```
RE = r1.reactions_from_stoich(stoich_coeff, add_odes=False)

# Modify component C due to its changing volume
RE['C'] += 0.02247311828 / (V * 210) * V_step

r1.add_odes(RE)
```

Note: You can still add additional ODEs to the ReactionModel afterwards. For example, if you need to add a volumetic flowrate (like the one influencing C above), this can be added in the usual manner using **add_ode**.

You may also generate the stoichiometric matrix from the finished system of ODEs using the **stoich_from_reactions** method. In order for this to work, you need to register the reactions using the **add_reaction** method. This becomes important for reactions involving unwanted contributions in the spectral data.

Note: Volume changes are automatically applied to the ODEs for all components. This follows the form of

$$-\frac{\dot{V}}{V} \cdot C_i$$

where V is the volume state, $\dot{V}$ is the volume's rate of change (its ODE), and C_i is the concentration of component i.

Note: You can disable the automatic generation of volume change terms in the settings:

```
r1.settings.general.add_volume_terms = False
```

1.4.4 Experimental Data

KIPET has several features that make it very simple to add experimental data to the ReactionModel. Before showing how to add data to the model, it is important to know how to format the data and where KIPET expects the data to be found.

KIPET expects the data file to be in the same directory, or a subdirectory thereof, as the python script containing the model. If this is the case, then using the relative path to the data file is acceptable in your script.

```
# Working directory

reaction.py
data/
    data_file.txt
```

The acceptable file in the example above is "data/data_file.txt". If the data is not in the project directory then the full path to the file should be used instead.

Data Formats

The data can be stored as a *.txt* or *.csv* file. For state data, the data should be formatted using the component or state name in the column header and the times in the index (the first columns) with no header. KIPET takes the file type into account and formats the data appropriately.

For example:

	A	B	C	D
1		A	B	C
2	0	0.001027028742248	0	1.26220046667191E-05
3	0.0333	0.001015414679356	2.20424950420788E-06	2.05558878076343E-05
4	0.0667	0.001006906145442	1.30414639175767E-05	1.76369281062045E-05
5	0.1	0.000992635293416	2.4680620670202E-05	1.37628033141861E-05
6	0.1334	0.00098201777305	3.4599256054874E-05	1.23810673181812E-05
7	0.1667	0.000975664716599	4.11158253447824E-05	1.17955865714225E-05
8	0.2	0.000960539560883	5.3551747983587E-05	1.09843376496413E-05
9	0.2334	0.000950302810967	6.04934532306774E-05	1.31516275743194E-05
10	0.2667	0.000941685492661	6.8398757115534E-05	1.05750176499706E-05
11	0.3001	0.00093460847914	7.53412578575117E-05	1.10390884072201E-05
12	0.3334	0.000928029690184	8.09167827666822E-05	9.03160249866784E-06
13	0.3667	0.000918578575377	8.54575497802487E-05	1.11497545099864E-05
14	0.4001	0.000908572538801	9.31849659099066E-05	1.30003145602247E-05

Fig. 1.7: How the data should be arranged for state data

If the data comes in the form of a *.txt* file, the data should be organized line for line like the following:

```
    time, component, measured value

0.0, A, 0.0010270287422477187
0.0, B, 0.0
0.0, C, 1.2622004666719102e-05
0.0333, A, 0.0010154146793560827
0.0333, B, 2.2042495042078835e-06
0.0333, C, 2.0555887807634343e-05
0.0667, A, 0.001006906145441944
0.0667, B, 1.3041463917576706e-05
0.0667, C, 1.7636928106204522e-05
0.1, A, 0.0009926352934163576
0.1, B, 2.4680620670202026e-05
0.1, C, 1.3762803314186124e-05
    etc...
```

Spectral data should have the first column contain the measurement times and the columns thereafter the wavelengths. See the image below for an example.

	A	B	C	D	E	F	G	H
1		180	181	182	183	184	185	186
2	0	0.0096	0.0196	0.0257	0.0288	0.0258	0.0191	0.0139
3	0.2	0.0089	0.0139	0.0184	0.0216	0.0181	0.0148	0.0087
4	0.4	0.0074	0.0110	0.0173	0.0169	0.0175	0.0124	0.0071
5	0.6	0.0022	0.0039	0.0047	0.0047	0.0043	0.0024	0.0036
6	0.8	0.0053	0.0085	0.0077	0.0128	0.0098	0.0070	0.0049
7	1	0.0038	0.0038	0.0074	0.0089	0.0095	0.0073	0.0038
8	1.2	0.0017	0.0032	0.0017	0.0023	0.0032	0.0032	0.0013
9	1.4	0.0042	0.0052	0.0060	0.0083	0.0089	0.0068	0.0043
10	1.6	0.0007	0.0020	0.0021	0.0043	0.0044	0.0031	0.0032
11	1.8	-0.0007	0.0019	0.0014	0.0017	0.0015	0.0016	0.0031
12	2	0.0016	-0.0017	-0.0004	-0.0011	-0.0008	-0.0011	0.0010
13	2.2	0.0049	0.0052	0.0070	0.0079	0.0071	0.0057	0.0065
14	2.4	0.0021	0.0019	0.0022	0.0048	0.0025	0.0035	0.0039
15	2.6	0.0053	0.0082	0.0104	0.0125	0.0107	0.0096	0.0072
16	2.8	-0.0016	-0.0010	-0.0004	-0.0012	-0.0010	0.0016	0.0020
17	3	0.0000	0.0012	0.0011	0.0026	0.0024	0.0018	0.0014
18	3.2	0.0052	0.0042	0.0045	0.0055	0.0032	0.0035	0.0048
19	3.4	0.0039	0.0050	0.0053	0.0053	0.0054	0.0029	0.0050
20	3.6	-0.0017	-0.0041	-0.0048	-0.0051	-0.0044	-0.0035	0.0003
21	3.8	0.0009	0.0032	0.0030	0.0039	0.0040	0.0041	0.0019

Fig. 1.8: How the data should be arranged for spectral data

If the data comes in the form of a *.txt* file, the data should be organized line for line like the following:

```
time, wavelength, measured value

0.0000, 1610.00, 0.074030
0.0000, 1620.00, 0.076191
0.0000, 1630.00, 0.077368
0.0000, 1640.00, 0.078412
0.0000, 1650.00, 0.083268
0.0000, 1660.00, 0.087972
```

(continues on next page)

(continued from previous page)

```
0.0000, 1670.00, 0.082916
0.0000, 1680.00, 0.084603
0.0000, 1690.00, 0.088627
0.0000, 1700.00, 0.089958
0.0000, 1710.00, 0.085465
etc...
```

Adding the Data

The previous cases assumed that you were loading a datafile directly into KIPET. If this is the case, adding data is as simple as

```
r1.add_data(file='data/data_file.txt')
```

using the same example as before.

If you wish to use your own data frame as data instead of loading directly from a file, this can be done by using the keyword argmument data:

```
r1.add_data(data=<your_dataframe_goes_here>)
```

Also, you can use the methods **read_data** and **write_data** to load and write data in the KIPET format. These methods are accessible at the top-level:

```
data_frame = kipet.read_data('filename')
data_frame = data_frame.iloc[:, ::10]
kipet.write_data('reduced_by_10_data.csv')
```

KIPET will automatically check if the entered components and states match with the column headers in the data added to the ReactionModel. It does not matter if the data is entered in before or after the components and states are declared. Once the data has been added to the ReactionModel, it can be accessed through the datasets attribute.

```
r1.datasets['name_of_the_dataset']
```

Here the name of the dataset is either provided as the first positional argument to **add_data** or will be automatically generated based on the type of data added. For example, concentration data added without a name is named 'C_data'. State data is given the name 'U_data'. The dataset class contains a handful of methods that can be used to visualize the data, remove negative values, and more.

Due to the difference in structure of the spectral data, it is not found in the datasets attribute, but rather in the spectra attribute. The spectra attribute of the ReactionModel class is really a SpectralHandler object that contains various preprocessing tools as well as a plotting tool specifically designed for spectral data.

See the add_data method in ReactionModel for more information pertaining to other arguments and methods related to data in KIPET. Also checkout the SpectraHandler class and the DataComponent class to learn more about these objects and how to manage data in KIPET.

1.4.5 Simulation

Once you have added all of the necessary components to the ReactionModel you are ready to perform simulation and/or parameter fitting. If you do not add any experimental data to the ReactionModel, you cannot proceed with parameter fitting and will only be able to perform simulations. In this case the simulator requires a start and an end time. In this case, the start time is generally zero and is not very important. The end time determines when the simulation will end and is therefore required. If you do provide experimental data with your simulation model, KIPET will automatically determine the end time based on the last measured data point. If you provide your own end time, this will override the time determined from the data.

```
r1.set_time(10)
```

In the example above the end time is set at 10 (units can be derived from the base time unit configured using the ReactionModel).

Once the model is complete with the start and end times, running the simulation is as simple as

```
r1.simulate()
```

The simulator in KIPET is based on a finite element by finite element approach to ensure robustness of the solution and a high chance of convergence.

Note: If you have trouble simulating a model it may be the case that it is too stiff. Try increasing the number of finite elements in the model until it converges.

```
r1.settings.collocation.nfe = <number of finite elements>
```

1.4.6 Parameter Fitting

The parameter fitting is also quite simple to use in KIPET. After the model is complete and includes experimental data, the parameter fitting can be started using the **run_opt** method.

```
r1.run_opt()
```

Depending on the type of problem, a series of steps begins so that the solution to the parameter fitting problem can be found. It begins with a simulation no different from performing a stand-alone simulation. This is done using the initial values to provide the model with good initial values for the variables not defined by the user. These include the concentration profiles, absorbance profiles, and other variables. This greatly increases the speed of convergence and reduces the chance of having it fail to solve. If the reaction data contains spectral data, the variance estimation stage follows the simulation. Here, also depending on the method used to estimate the variances, the variance of each component is predicted from the spectral data and the model structure. After these variances are known, the parameter fitting may proceed. If only concentration data is present, the variances need to be provided to the model before the parameter fitting begins. Thus, problems with only concentration data jump straight to parameter fitting and do not require a variance estimation step. After the parameter estimation is complete, the results are stored in a ResultsObject that can be reached using the results attribute of the ReactionModel instance. All relevant variables are found here alongside their optimal trajectories in convenient data forms dependent upon the dimensionality of the data.

Also, once the results are available, there are many plotting tools that can be used to plot the various results obtained. This is accessed using the **plot** method of the ReactionModel class and takes the variable name in the model as the parameter. If no parameter is passed to **plot**, all plots related to the model are generated.

1.4.7 Plotting Results

KIPET provides robust plotting methods that make it easy to display the results from simulation and parameter fitting problems. Plots are generated using the Plotly package and are saved as both HTML and SVG file types. The plot methods can be accessed simply by using the **plot** method. All plots are stored in a folder called "charts" that will be created in the working directory.

```
r1.plot() # plots all related charts
r1.plot('Z') # plots all concentration charts
r1.plot('A') # plots the component A
r1.plot('V') # plots the volume
# and so on
```

1.4.8 Settings

If you happen to have used an earlier version of KIPET, you may have noticed that the user was responsible for entering in various options for the variance estimator, parameter estimation, multiple experiments estimator, and so on. In the latest version of KIPET, many of the options are maintained in the background with usually good default values for most problems. However, if you would like to change the default settings, you are free to do so. This can be done by accessing the Settings object through the settings attribute of the ReactionModel instance.

For example, you can change the number of collocation points and the number of finite elements by the following:

```
r1.settings.collocation.ncp = 3
r1.settings.collocation.nfe = 100
```

To see all options available, simply type r1.settings into the command prompt. The name in parathenses after the settings section title (i.e. General Settings (general)) is the name used to access this setting. For example, in the example above we change the collocation setting by accessing settings.collocation.ncp and settings.collocation.nfe, as can be seen below. In this way, the user can modify any of the settings using simple dot notation.

```
>> r1.settings

    General Settings (general):
           confidence : 1
       initialize_pe : True
      no_user_scaling : True
     scale_parameters : False
             scale_pe : True
      scale_variances : False
     simulation_times : None

    Unit Settings (units):
        concentration : M
                 time : hr
               volume : L

    Collocation Settings (collocation):
               method : dae.collocation
                  ncp : 3
                  nfe : 60
               scheme : LAGRANGE-RADAU
```

(continues on next page)

(continued from previous page)

```
    Simulation Settings (simulator):
               solver : ipopt
                  tee : False
         solver_opts : {}

# and many more...
```

If you would like to change the default values permanently, you can do this by changing the settings.yml file in the kipet directory.

1.5 Examples

This section includes several examples showing many of the features included in KIPET. If you are new to KIPET, it is recommended to first visit the Getting Started with KIPET section to learn the basics. Once you are familiar with the basic structure of KIPET, the following examples can be used as starting points for various types of project needs.

1.5.1 Introduction to Simulation

Files Ex_1_ode_sim.py

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.

$$\mathrm{A} \xrightarrow{r_A} \mathrm{B} \xrightarrow{r_B} \mathrm{C}$$

$$\mathrm{r}_A = k_1 C_A$$
$$\mathrm{r}_B = k_2 C_B$$

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.

```
# Create the ReactionModel instance
r1 = kipet.ReactionModel('reaction-1')

# Change the desired time basis here (if different from default)
r1.unit_base.time = 's'

# Add the model parameters
k1 = r1.parameter('k1', value=2, units='1/s')
k2 = r1.parameter('k2', value=0.2, units='1/s')

# Declare the components and give the initial values
A = r1.component('A', value=1.0, units='M')
B = r1.component('B', value=0.0, units='M')
C = r1.component('C', value=0.0, units='M')

# Input the reactions as expressions
rA = r1.add_reaction('rA', k1*A)
rB = r1.add_reaction('rB', k2*B)

# Input the ODEs
```

(continues on next page)

(continued from previous page)

```
r1.add_ode('A', -rA )
r1.add_ode('B', rA - rB )
r1.add_ode('C', rB )

# Option to check the units of your models
r1.check_model_units(display=True)

# Add dosing points
r1.add_dosing_point('A', 3, 0.3)

# Simulations require a time span
r1.set_time(10)

# Change some of the default settings
r1.settings.collocation.ncp = 3
r1.settings.collocation.nfe = 50

# Simulate
r1.simulate()

# Create plots
r1.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.

```
import kipet
```

The kipet package contains all of the methods necessary to use KIPET. The next step is to create an instance of the ReactionModel class. Note that the reaction requires a name as the first argument.

```
r1 = ReactionModel('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 **parameter** method, as seen in the current example where there are two parameters:

```
k1 = r1.parameter('k1', value=2)
k2 = r1.parameter('k2', value=0.2)
```

Since our system has three components, A, B, and C, these need to be declared as well. Each component requires at a minimum a name. For simulations, an initial value for each of the components is also required.

```
A = r1.component('A', value=1)
B = r1.component('B', value=0.0)
C = r1.component('C', value=0.0)
```

The next step is to provide the equations needed to define the reaction kinetics. The reaction kinetic rules are placed into the model using the **add_reaction** method. Please note that KIPET requires that each declared component has its own expression. Once the reactions have been declared, the ODEs for each component can be constructed.

```
# Define explicit system of ODEs
rA = r1.add_reaction('rA', k1*A )
```

(continues on next page)

(continued from previous page)

```
rB = r1.add_reaction('rB', k2*B)

# Add the ODEs to the model
r1.add_ode('A', -rA)
r1.add_ode('B', rA - rB)
r1.add_ode('C', rB)
```

At this point we have provided KIPET with a reaction model, component information, and parameter data. The start time is always set to zero so only the duration of the simulation is needed. This can be set using the **set_time** method. As we will see in the parameter estimation problems, explicitly providing start and end times is not necessary if experimental data is provided.

```
r1.set_time(10)
```

After this we are ready to simulate using the **simulate** method. The results are then accessible using the **results** attribute. This attribute points to an instance of the ResultsObject class. The most basic plotting tool can be accessed using the **plot** method of the ReactionModel instance.

```
r1.simulate()
r1.plot()
```

The results are then presented in a new browser tab using Plotly similar to the following figure. Figures are also saved as SVG files in the same directory.

Fig. 1.9: Plot obtained from tutorial example 1

1.5.2 Introduction to 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 the previous example, except in this case we use a simulated spectral data set as our input D-matrix. This example, while not too computationally complex provides a good overview of all the capabilities of KIPET. So, again we are considering the following:

$$\mathrm{A} \xrightarrow{r_A} \mathrm{B} \xrightarrow{r_B} \mathrm{C}$$

$$\mathrm{r}_A = k_1 C_A$$
$$\mathrm{r}_B = k_2 C_B$$

The full code for this example:

```
import kipet

r1 = kipet.ReactionModel('reaction-1')

# Add the model parameters
k1 = r1.parameter('k1', value=0.3, bounds=(0.0, 5.0))
k2 = r1.parameter('k2', value=1.4, bounds=(0.0, 3.0))
```

(continues on next page)

(continued from previous page)

```
# Declare the components and give the initial values
A = r1.component('A', value=1)
B = r1.component('B', value=0.0)
C = r1.component('C', value=0.0)

# Input data
file_name = 'data/Dij.txt'
r1.add_data(category='spectral', file=file_name)

# Input the reactions as expressions
rA = r1.add_reaction('rA', k1*A)
rB = r1.add_reaction('rB', k2*B)

# Input the ODEs
r1.add_ode('A', -rA )
r1.add_ode('B', rA - rB )
r1.add_ode('C', rB )

# Optinal bounds on the S profiles
r1.bound_profile(var='S', bounds=(0, 10))

# Change some of the default settings
r1.settings.collocation.ncp = 1
r1.settings.collocation.nfe = 60
r1.settings.parameter_estimator.tee = True

# Parameter fitting
r1.run_opt()

# Plot results
r1.plot()
```

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

Reading data

Firstly we will need to input our D-matrix that contains the spectral data. More notes on the structure of these data files and which file types are permitted are found *here*. 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_data(category='spectral', file='data/Dij.txt')
```

Spectra data is handled in a different manner than state data. The ReactionModel object has a spectra attribute that is a SpectralData object that allows for various pre-processing techniques to be applied to the spectral data before the parameter fitting.

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

```
r1.bound_profile(var='S', bounds=(0, 10)
```

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.

```
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. This can be accessed using the **decrease_wavelengths** method of the spectra attribute. This method takes an integer as the first (and required) parameter.

```
r1.spectra.decrease_wavelengths(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. For more information about settings in KIPET, see *here*.

```
r1.settings
```

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 in the IPOPT documentation.

Solver settings can be set in the following manner:

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

Variance Estimation and Parameter Fitting

Previously the variance estimation step was required to be set up by the user. This is now handled in the background by KIPET for you. You can change the way in which the variances are calculated by altering the settings.

To solve the problem, simply run the following:

```
r1.run_opt()
```

The results and plots can now be displayed.

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

The results will then be shown as:

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

Providing us with the following plots:

Fig. 1.10: Concentration profile results from tutorial example 2

Fig. 1.11: Pure component absorbance profiles (S) result from tutorial example 2

1.5.3 Simulating 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 this example a problem 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** method.

The system of equations is:

$$
\begin{aligned}
k1 &= 1.25e^{\frac{9500}{1.987}(\frac{1}{320}-\frac{1}{T})} && (1.7)\\
k2 &= 0.08e^{\frac{7000}{1.987}(\frac{1}{290}-\frac{1}{T})} && (1.8)\\
r_A &= -(k_1 A) && (1.9)\\
r_B &= 0.5k_1A - (k_2 B) && (1.10)\\
r_C &= (k_2 B) && (1.11)\\
C_{A0} &= 1.2 && (1.12)\\
V_0 &= 1.0 && (1.13)\\
T_1 &= 35000(298 - T) && (1.14)\\
T_2 &= 4 \cdot 240 \cdot 30(T - 305) && (1.15)\\
T_3 &= V(6500k_1A - 8000(k_2 B)) && (1.16)\\
D_{en} &= (30A + 60B + 20C)V + 0.5T && (1.17)\\
& && (1.18)
\end{aligned}
$$

Using these expressions, the ODEs for this example reaction are:

$$
\begin{aligned}
\dot{A} &= r_A + (C_{A0} - A)/V && (1.19)\\
\dot{B} &= r_B - B(V_0/V) && (1.20)\\
\dot{C} &= r_C - C(V_0/V) && (1.21)\\
\dot{T} &= (T_1 + T_2 + T_3)/(1.22e) && (1.22)\\
\dot{V} &= 1.23 && (1.23)
\end{aligned}
$$

At this time, modeling using certain expressions (like 'exp' in the following expressions) requires importing the functions from pyomo.core.

```
import kipet

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

r1 = kipet.ReactionModel('reaction-1')

# Declare the components and give the initial values
A = r1.component('A', value=1.0)
```

(continues on next page)

(continued from previous page)

```
B = r1.component('B', value=0.0)
C = r1.component('C', value=0.0)

# Declare the complementary states and their initial values
T = r1.state('T', value=290, description='Temperature')
V = r1.state('V', value=100, description='Volumne')
```

Similar to components, each complementary state will require an ODE to accompany it. In the case of this tutorial example, the following ODEs are defined:

```
# Define the expressions - note that expression method is not used!
k1 = 1.25*exp((9500/1.987)*(1/320.0 - 1/T))
k2 = 0.08*exp((7000/1.987)*(1/290.0 - 1/T))

ra = -k1*A
rb = 0.5*k1*A - k2*B
rc = 3*k2*B

cao = 4.0
vo = 240
T1 = 35000*(298 - T)
T2 = 4*240*30.0*(T-305.0)
T3 = V*(6500.0*k1*A - 8000.0*k2*B)
Den = (30*A + 60*B + 20*C)*V + 3500.0

# Add ODEs
r1.add_ode('A', ra + (cao - A)/V )
r1.add_ode('B', rb - B*vo/V )
r1.add_ode('C', rc - C*vo/V )
r1.add_ode('T', (T1 + T2 + T3)/Den )
r1.add_ode('V', vo )

# Simulation requires a time span
r1.set_time(2.0)

# Change some of the default settings
r1.settings.collocation.nfe = 20
r1.settings.collocation.ncp = 1

# Simulation
r1.simulate()

# Create plots
r1.plot()
```

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.

Fig. 1.12: Concentration profiles from Tutorial 3

Fig. 1.13: Temperature profile from Tutorial 3

1.5.4 Simulation of Advanced Reaction System with Algebraic equations (I)

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:

$$\mathrm{SA} + \mathrm{AA} \xrightarrow{k_1} \mathrm{ASA} + \mathrm{HA} \tag{1.24}$$

$$\mathrm{ASA} + \mathrm{AA} \xrightarrow{k_2} \mathrm{ASAA} + \mathrm{HA} \tag{1.25}$$

$$\mathrm{ASAA} + \mathrm{H_2O} \xrightarrow{k_3} \mathrm{ASA} + \mathrm{ASA} \tag{1.26}$$

$$\mathrm{AA} + \mathrm{H_2O} \xrightarrow{k_4} 2\,\mathrm{HA} \tag{1.27}$$

$$\mathrm{SA}(s) \xrightarrow{k_d} \mathrm{SA} \tag{1.28}$$

$$\mathrm{ASA}(l) \xrightarrow{k_c} \mathrm{ASA}(s) \tag{1.29}$$

With rate laws:

$$r_1 = k_1 c_{SA}(t) c_{AA}(t) \tag{1.30}$$

$$r_2 = k_2 c_{ASA}(t) c_{AA}(t) \tag{1.31}$$

$$r_3 = k_3 c_{ASAA}(t) c_{H_2O}(t) \tag{1.32}$$

$$r_4 = k_4 c_{AA}(t) c_{H_2O}(t) \tag{1.33}$$

$$r_d = \begin{cases} k_d (c_{SA}^{sat}(T) - c_{SA}(t))^d, & \text{if } m_{SA} \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{1.34}$$

$$r_c = k_c (\max(c_{ASA}(t) - c_{ASA}^{sat}(T), 0))^c \tag{1.35}$$

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

$$\dot{m}_{SA} = -M_{SA} V r_d \tag{1.36}$$

$$\dot{c}_{SA} = r_d - r_1 - \frac{\dot{V}}{V} c_{SA} \tag{1.37}$$

$$\dot{c}_{AA} = -r_1 - r_2 - r_4 - \frac{\dot{V}}{V} c_{AA} \tag{1.38}$$

$$\dot{c}_{HA} = r_1 + r_2 + r_3 + 2 \cdot r_4 - \frac{\dot{V}}{V} c_{HA} \tag{1.39}$$

$$\dot{m}_{ASA} = M_{ASA} V r_g \tag{1.40}$$

$$\dot{c}_{ASA} = r_1 - r_2 + r_3 - r_g - \frac{\dot{V}}{V} c_{ASA} \tag{1.41}$$

$$\dot{c}_{ASAA} = r_2 - r_3 - \frac{\dot{V}}{V} c_{ASAA} \tag{1.42}$$

$$\dot{c}_{H_2O} = -r_3 - r_4 + \frac{f}{V} c_{H_2O}^{in} - \frac{\dot{V}}{V} c_{H_2O} \tag{1.43}$$

$$\dot{V} = V \sum_{i=1}^{ns} \nu_i \left(\sum_{j=1}^{4} \gamma_{i,j} r_j + \gamma_{i,d} r_d + \gamma_{i,c} r_c + \epsilon_i \frac{f}{V} c_{H_2O}^{in} \right) \tag{1.44}$$

From these two sets of equations, we can derive the algebraic expressions for the reactions and then construct the ODEs for the components and other states.

```
from pyomo.environ import exp
import kipet

rm = kipet.ReactionModel('reaction-1')

# Components
SA = rm.component('SA', value=1.0714, description='Salicitilc acid')
AA = rm.component('AA', value=9.3828, description='Acetic anhydride')
ASA = rm.component('ASA', value=0.0177, description='Acetylsalicylic acid')
HA = rm.component('HA', value=0.0177, description='Acetic acid')
ASAA = rm.component('ASAA', value=0.000015, description='Acetylsalicylic anhydride')
H2O = rm.component('H2O', value=0.0, description='Water')

# Parameters
k0 = rm.parameter('k0', value=0.0360309)
k1 = rm.parameter('k1', value=0.1596062)
k2 = rm.parameter('k2', value=6.8032345)
k3 = rm.parameter('k3', value=1.8028763)
kd = rm.parameter('ks', value=7.1108682)
kc = rm.parameter('kc', value=0.7566864)
Csa = rm.parameter('Csa',value=2.06269996)
```

In this example we need to declare new sets of states in addition to our components and parameters, as with the previous example.

```
# Additional state variables
V = rm.state('V', value=0.0202)
Masa = rm.state('Masa', value=0.0)
Msa = rm.state('Msa', value=9.537)

# Fixed states (data is provided using the data keyword argument)
f = rm.fixed_state('f', description='flow f', data='traj')
Csat = rm.fixed_state('Csat', description='C saturation', data='traj')
```

Note: When using fixed states the FESimulator cannot be used! The simulator needs to be set to 'dae.collocation' in the settings. This is due to the fixed algebraic trajectories not being supported at this time.

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

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

epsilon = dict()
```

(continues on next page)

(continued from previous page)

```
epsilon['SA']= 0.0
epsilon['AA']= 0.0
epsilon['ASA']= 0.0
epsilon['HA']= 0.0
epsilon['ASAA']= 0.0
epsilon['H2O']= 1.0

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

At this point we can add the data to the model. 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).

```
filename = 'data/extra_states.txt'
rm.add_data('traj', category='trajectory', file=filename)

filename = 'data/concentrations.txt'
rm.add_data('conc', category='trajectory', file=filename)

filename = 'data/init_Z.csv'
rm.add_data('init_Z', category='trajectory', file=filename)

filename = 'data/init_X.csv'
rm.add_data('init_X', category='trajectory', file=filename)

filename = 'data/init_Y.csv'
rm.add_data('init_Y', category='trajectory', file=filename)
```

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

```
# Reaction rates
r0 = rm.add_reaction('r0', k0*SA*AA, description='Reaction 0')
r1 = rm.add_reaction('r1', k1*ASA*AA, description='Reaction 1' )
r2 = rm.add_reaction('r2', k2*ASAA*H2O, description='Reaction 2' )
r3 = rm.add_reaction('r3', k3*AA*H2O, description='Reaction 3')

step = 1/(1 + exp(-Msa/1e-4))
r4 = rm.add_reaction('r4', kd*(Csa - SA + 1e-6)**1.90*step, description='Reaction 4' )

diff = ASA - Csat
r5 = rm.add_reaction('r5', 0.3950206559*kc*(diff+((diff)**2+1e-6)**0.5)**1.34,␣
↪description='Reaction 5' )
```

Since the stoichiometric coefficients and the reaction equations are known, the component ODEs can be built automatically. Since the ODEs need to be augmented with the volume changes, the ODEs are returned from the reactions_from_stoich method. Once finished with the ODEs, they need to be added to the model in the usual way using add_ode.

```
# Generate the ODEs for the reactions based on the stoichiometric matrix
# Since we need to modfiy the ODEs, add_odes should be False
odes = rm.reactions_from_stoich(gammas, add_odes=False)

v_sum_float = 0
Cin = 39.1

# Build expression for the volume
for com in rm.components.names:
    v_sum_float += partial_vol[com] * (odes[com] + epsilon[com]*f/V*Cin)

v_sum = rm.add_expression('v_sum', v_sum_float, description='Volume Sum')

# If calling a component (such as A or B) in a loop, use the pyomo_var attribute
# Add ODEs for the components
for com in rm.components.names:
    rm.add_ode(com, odes[com] + epsilon[com]*f/V*Cin - v_sum*rm.components[com].pyomo_
↪var)

# Add ODEs for complementary states
rm.add_ode('V', V*v_sum )
rm.add_ode('Masa', 180.157*V*r5 )
rm.add_ode('Msa', -138.121*V*r4 )
```

At this point the model is almost ready and only the final details are needed before simulation.

```
# Simulations require a time span
rm.set_time(210.5257)

# Settings
rm.settings.collocation.nfe = 100
    rm.settings.simulator.method = 'dae.collocation'
rm.settings.simulator.solver_opts.update({'halt_on_ampl_error' :'yes'})

# Initialize the model variables with the provided data
rm.initialize_from_trajectory('Z', 'init_Z')
rm.initialize_from_trajectory('X', 'init_X')
rm.initialize_from_trajectory('Y', 'init_Y')

# Run the simulation
rm.simulate()

    # Plot the results
        rm.plot('Z')
```

Fig. 1.14: Concentration profiles from Tutorial 4.

1.5.5 Simulation of Advanced Reaction System with Algebraic equations (II)

Files Ex_5_sim_fe_by_fe.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 the previous example. In this tutorial, the following reaction system is simulated.

$$\begin{aligned}
\mathrm{AH} + \mathrm{B} &\xrightarrow{r_1} \mathrm{A^-} + \mathrm{BH^+} && (1.45)\\
\mathrm{A^-} + \mathrm{C} &\xrightarrow{r_2} \mathrm{AC^-} && (1.46)\\
\mathrm{AC^-} &\xrightarrow{r_{-2}} \mathrm{A^-} + \mathrm{C} && (1.47)\\
\mathrm{AC^-} + \mathrm{AH} &\xrightarrow{r_3} \mathrm{P} + \mathrm{A^-} && (1.48)\\
\mathrm{AC^-} + \mathrm{BH^+} &\xrightarrow{r_4} \mathrm{P} + \mathrm{AH} + \mathrm{B} && (1.49)\\
& && (1.50)
\end{aligned}$$

$$\begin{aligned}
r_1 &= k_1 C_{AH} C_B && (1.51)\\
r_2 &= k_2 C_{A^-} C_C && (1.52)\\
r_{-2} &= k_{-2} C_{AC^-} && (1.53)\\
r_3 &= k_3 C_{AC^-} C_{AH} && (1.54)\\
r_4 &= k_4 C_{AC^-} C_{BH^+} && (1.55)
\end{aligned}$$

Which is represented by the following ODE system:

$$\frac{dV}{dt} = \begin{cases} \text{const flowrate}, & t 3.5h \\ 0, & t \geq 3.5h \end{cases} \quad (1.56)$$

$$\begin{aligned}
\frac{dC_{AH}}{dt} &= -r_1 - r_3 - \frac{\dot{V}}{V} C_{AH} && (1.57)\\
\frac{dC_B}{dt} &= -r_1 + r_4 - \frac{\dot{V}}{V} C_B && (1.58)\\
\frac{dC_{A^-}}{dt} &= r_1 - r_2 + r_{-2} + r_3 - \frac{\dot{V}}{V} C_{A^-} && (1.59)\\
\frac{dC_{BH^+}}{dt} &= r_1 - r_4 - \frac{\dot{V}}{V} C_{BH^+} && (1.60)\\
\frac{dC_C}{dt} &= -r_2 + r_{-2} - \frac{\dot{V}}{V} C_C + \begin{cases} m_{C_{add}}/V/3.5, & t 3.5h \\ 0, & t \geq 3.5h \end{cases} && (1.61)\\
\frac{dC_{AC^-}}{dt} &= r_2 - r_{-2} - r_3 - r_4 - \frac{\dot{V}}{V} C_{AC^-} && (1.62)\\
\frac{dC_P}{dt} &= r_3 + r_4 - \frac{\dot{V}}{V} C_P && (1.63)
\end{aligned}$$

```
import kipet

    r1 = kipet.ReactionModel('reaction-1')

# Components
AH = r1.component('AH', value= 0.395555)
B = r1.component('B', value= 0.0351202)
C = r1.component('C', value= 0.0)
```

(continues on next page)

(continued from previous page)

```
BHp = r1.component('BHp', value= 0.0)
Am = r1.component('Am', value= 0.0)
ACm = r1.component('ACm', value= 0.0)
P = r1.component('P', value= 0.0)

# Parameters
k0 = r1.parameter('k0', value=49.7796)
k1 = r1.parameter('k1', value=8.93156)
k2 = r1.parameter('k2', value=1.31765)
k3 = r1.parameter('k3', value=0.31087)
k4 = r1.parameter('k4', value=3.87809)

# States
V = r1.volume(value=0.0629418, units='L')

# Stoichiometric coefficients
stoich_coeff = dict()
stoich_coeff['AH'] = [-1, 0, 0, -1, 0]
stoich_coeff['B'] = [-1, 0, 0, 0, 1]
stoich_coeff['C'] = [0, -1, 1, 0, 0]
stoich_coeff['BHp'] = [1, 0, 0, 0, -1]
stoich_coeff['Am'] = [1, -1, 1, 1, 0]
stoich_coeff['ACm'] = [0, 1, -1, -1, -1]
stoich_coeff['P'] = [0, 0, 0, 1, 1]

V_step = r1.step('V_step', time=210, fixed=True, switch='off')
V_flow = r1.constant('V_flow', value=7.27609e-5)

y0 = r1.add_reaction('y0', k0*AH*B, description='Reaction 0')
y1 = r1.add_reaction('y1', k1*Am*C, description='Reaction 1')
y2 = r1.add_reaction('y2', k2*ACm, description='Reaction 2')
y3 = r1.add_reaction('y3', k3*ACm*AH, description='Reaction 3')
y4 = r1.add_reaction('y4', k4*ACm*BHp, description='Reaction 4')

RE = r1.reactions_from_stoich(stoich_coeff, add_odes=False)
# Modify component C
RE['C'] += 0.02247311828 / (V * 210) * V_step

# ODEs - volume change terms added automatically
r1.add_odes(RE)
r1.add_ode('V', V_flow*V_step)
```

The dosing points need to be entered in using the following syntax: ('component name', time, amount). See the following code as an example. You can add as many such dosing points as needed.

```
# Add dosing points (as many as you want in this format)
# ('component_name', time, conc=(value, units), volume=(amount, units))
r1.add_dosing_point('AH', time=100, conc=(1.3, 'M'), vol=(20, 'mL'))
```

The simulation is now ready.

```
r1.set_time(600)
```

(continues on next page)

(continued from previous page)

```
r1.settings.collocation.nfe = 40
r1.settings.simulator.method = 'fe'

r1.simulate()

if with_plots:
    r1.plot('Z')
```

Fig. 1.15: Concentration profile of solution to Tutorial 5

1.5.6 Reaction Systems with Known Non-Absorbing Species

Files Ex_6_non_absorbing.py

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.

```
r1.component('C', value=0.0, absorbing=False)
```

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

```
import kipet

r1 = kipet.ReactionModel('reaction-1')

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

# Declare the components and give the initial values
A = r1.component('A', value=1)
B = r1.component('B', value=0.0, known=True, bounds=(1e-8, 1e-4))
C = r1.component('C', value=0.0, absorbing=False)

# Add the data
r1.add_data(category='spectral', file='example_data/Dij.txt')

# define explicit system of ODEs
rA = r1.add_reaction('rA', k1*A, description='Reaction A' )
rB = r1.add_reaction('rB', k2*B, description='Reaction B' )

# Use the stoichiometry to build the reaction network:
stoich_data = {'rA': [-1, 1, 0],
               'rB': [0, -1, 1]}

r1.reactions_from_stoich(stoich_data)

# Settings
```

(continues on next page)

(continued from previous page)

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

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

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

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

Fig. 1.16: Concentration profile of solution to Tutorial 6

Fig. 1.17: Absorbance profiles in Tutorial 6

1.5.7 Parameter Estimation Using Concentration Data

Files

Ex_7_concentration_heterogeneous_data.py
Ex_7_concentration_input.py
Ex_7_binary_reaction.py

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.

```
import kipet

r1 = kipet.ReactionModel('reaction-1')

# Add the model parameters
k1 = r1.parameter('k1', value=2.0, bounds=(0.0, 5.0))
k2 = r1.parameter('k2', value=0.2, bounds=(0.0, 2.0), fixed=False)

# Declare the components and give the initial values
A = r1.component('A', value=0.001, variance=1e-10, known=True, bounds=(0.0, 3))
B = r1.component('B', value=0.0, variance=1e-11)
C = r1.component('C', value=0.0, variance=1e-8)
```

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. In the case that data is added before, KIPET will match up the data with components before building the model in TemplateBuilder.

In the following, the data is first loaded and only every 10th data point is used in parameter fitting.

```
# Load the data and reduce the number of data points
filename = 'data/Ex_1_C_data.txt'
full_data = kipet.read_data(filename)
r1.add_data(data=full_data.iloc[::10, :], remove_negatives=True)

# Define the reaction model
r1.add_ode('A', -k1 * A )
r1.add_ode('B', k1 * A - k2 * B )
r1.add_ode('C', k2 * B )

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

# Run KIPET
r1.run_opt()

# Display the results
r1.results.show_parameters
r1.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:

```
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.

```
r1.settings.parameter_estimator.solver = 'k_aug'
```

The covariance matrix of the parameters can then be displayed:

```
r1.results.parameter_covariance
```

Fig. 1.18: Concentration profile results using concentration data (concentration input)

In Ex_7_binary_reaction.py the same problem is solved. The only difference is that measurements occur before the reaction begins. Although this may not be realistic, its purpose is to show how parameter fitting can be performed when the exact time the reaction starts is unknown. This is done use approximated binary decision variables that are still continuous variables. Thus, no integer decisions neede to be made.

You can very easily include such decision variables using the step method of the ReactionModel class. This method takes several arguments including the variable name, the time where the step changes, whether this time is fixed, and whether the step is turning on or off (going to one or to zero). If fixed is False, then the time argument is used as the initial value.

```
# Use step functions to turn on the reactions
b1 = r1.step('b1', time=2, fixed=False, switch='on')

# If desired, you can use different times for each reaction
# b2 = r1.step('b2', time=2.1, fixed=True, switch='on')
```

(continues on next page)

(continued from previous page)

```
rA = b1*(k1*A)
rB = b1*(k2*B)
```

Fig. 1.19: Concentration profile with unknown reaction start (binary_reaction)

Fig. 1.20: Profile of step function used to model the reaction start (binary reaction)

1.5.8 Variance and Parameter Estimation with Time-Dependent Inputs

Coming soon

1.5.9 Interfering Species and Fixed Absorbances

This tutorial is under construction! Check back shortly.

1.5.10 Parameter Estimability Analysis

Files

Ex_8_estimability.py
Ex_9_estimability_with_problem_gen.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.

```
import kipet

r1 = kipet.ReactionModel('reaction-1')

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

# Declare the components and give the initial values
A = r1.component('A', value=0.3)
B = r1.component('B', value=0.0)
C = r1.component('C', value=0.0)
D = r1.component('D', value=0.01)
E = r1.component('E', value=0.0)

filename = 'data/new_estim_problem_conc.csv'
r1.add_data('C_frame', file=filename)
```

(continues on next page)

(continued from previous page)

```
r1.add_ode('A', -k1*A - k4*A )
r1.add_ode('B',  k1*A - k2*B - k3*B )
r1.add_ode('C',  k2*B - k4*C )
r1.add_ode('D',  k4*A - k3*D )
r1.add_ode('E',  k3*B )

r1.set_times(0, 20)

param_uncertainties = {'k1':0.09,'k2':0.01,'k3':0.02,'k4':0.5}
# sigmas, as before, represent the variances in regard to component
sigmas = {'A':1e-10,'B':1e-10,'C':1e-11, 'D':1e-11,'E':1e-11,'device':3e-9}
# measurement scaling
meas_uncertainty = 0.05

params_fit, params_fix = r1.analyze_parameters(method='yao',
                              parameter_uncertainties=param_uncertainties,
                              meas_uncertainty=meas_uncertainty,
                              sigmas=sigmas)
```

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:

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:

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 this example.

1.5.11 Using Wavelength Selection Tools

Files Ex_10_estimation_lof_correlation_subset_selection.py

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.

```
import kipet

r1 = kipet.ReactionModel('reaction-1')

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

# Declare the components and give the initial values
A = r1.component('A', value=1)
B = r1.component('B', value=0.0)
C = r1.component('C', value=0.0)

# Use this function to replace the old filename set-up
r1.add_data('D_frame', category='spectral', file='data/Dij.txt')

# Define explicit system of ODEs
rA = r1.add_reaction('rA', k1*A)
rB = r1.add_reaction('rB', k2*B)

# Add the ODEs to the model
r1.add_ode('A', -rA)
r1.add_ode('B', rA - rB)
r1.add_ode('C', rB)

    # Place a bound on the species' absorption profiles (if known)
r1.bound_profile(var='S', bounds=(0, 200))

# Settings
r1.settings.collocation.ncp = 3
r1.settings.collocation.nfe = 60
# This needs to be set to False to keep the reduced hessian output surpressed
r1.settings.parameter_estimator.tee = False
r1.settings.parameter_estimator.solver = 'ipopt_sens'
r1.settings.solver.mu_init = 1e-4

r1.run_opt()

# Display the results
r1.results.show_parameters
r1.plot()
```

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

```
lof = r1.lack_of_fit()
    >> The lack of fit is  1.3914660989504697  %
```

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

```
correlations = r1.wavelength_correlation(corr_plot=True)
```

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

Fig. 1.21: Wavelength correlations

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:

```
r1.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 wavelngths and their correlations obtained above), lof (the lack of fit from the full parameter estimation problem, i.e. where all the wavelengths are selected), followed by the nfe (number of finite elements), ncp (number of collocation points), and the sigmas (variances from VarianceEstimator). These are the required arguments for the function. The outputs are as follows:

```
>> r1.run_lof_analysis()

When wavelengths of less than  0.00 correlation are removed
The lack of fit is:  1.391466 %
When wavelengths of less than  0.20 correlation are removed
The lack of fit is:  1.407669 %
When wavelengths of less than  0.40 correlation are removed
The lack of fit is:  1.412721 %
When wavelengths of less than  0.60 correlation are removed
The lack of fit is:  1.431436 %
When wavelengths of less than  0.80 correlation are removed
The lack of fit is:  1.684903 %
```

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:

```
>> r1.run_lof_analysis(step_size = 0.01, search_range = (0, 0.12))

When wavelengths of less than 0.000 correlation are removed
The lack of fit is: 1.391466 %
When wavelengths of less than 0.010 correlation are removed
The lack of fit is: 1.391466 %
When wavelengths of less than 0.020 correlation are removed
The lack of fit is: 1.391466 %
When wavelengths of less than 0.030 correlation are removed
The lack of fit is: 1.387124 %
```

(continues on next page)

(continued from previous page)

```
When wavelengths of less than 0.040 correlation are removed
The lack of fit is: 1.387124 %
When wavelengths of less than 0.050 correlation are removed
The lack of fit is: 1.388986 %
When wavelengths of less than 0.060 correlation are removed
The lack of fit is: 1.393768 %
When wavelengths of less than 0.070 correlation are removed
The lack of fit is: 1.390587 %
When wavelengths of less than 0.080 correlation are removed
The lack of fit is: 1.388974 %
When wavelengths of less than 0.090 correlation are removed
The lack of fit is: 1.391290 %
When wavelengths of less than 0.100 correlation are removed
The lack of fit is: 1.391290 %
When wavelengths of less than 0.110 correlation are removed
The lack of fit is: 1.391290 %
When wavelengths of less than 0.120 correlation are removed
The lack of fit is: 1.391290 %
```

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:

```
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:

```
subset_results = r1.run_opt_with_subset_lambdas(subset)

# Display the new results
subset_results.show_parameters
```

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

```
The estimated parameters are:
k1 0.3387633623377142
k2 1.4770816755838945
```

Fig. 1.22: Resulting concentration profiles

Fig. 1.23: Absorbance profiles

1.5.12 Parameter Estimation Using 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 than one ReactionModel instance in its models attibute. 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 example considered here involves two concentration datasets for the same reaction. The second reaction model uses only a sampling on points taken from the first dataset but with added noise.

```
import kipet

    lab = kipet.ReactionSet()

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

# Add the parameters
k1 = r1.parameter('k1', value=1.0, bounds=(0.0, 10.0))
k2 = r1.parameter('k2', value=0.224, bounds=(0.0, 10.0))

# Declare the components and give the initial values
A = r1.component('A', value=1.0e-3)
B = r1.component('B', value=0.0)
C = r1.component('C', value=0.0)

# define explicit system of ODEs
rA = k1*A
rB = k2*B

# Define the reaction model
r1.add_ode('A', -rA )
r1.add_ode('B', rA - rB )
r1.add_ode('C', rB )

# Add the dataset for the first model
r1.add_data('C_data', file='data/Ex_1_C_data.txt')

# Add the known variances
r1.variances = {'A':1e-10,'B':1e-10,'C':1e-10}

# Declare the second model (based on first model)
r2 = lab.new_reaction(name='reaction-2', model=r1)
```

(continues on next page)

(continued from previous page)

```
# Add the dataset for the first model
noised_data = kipet.add_noise_to_data(r1.datasets['C_data'].data, 0.0001)
r2.add_data('C_data', data=noised_data[::10])

# Add the known variances
r2.components.update('variance', {'A':1e-4,'B':1e-4,'C':1e-4})

# Create the MultipleExperimentsEstimator and perform the parameter fitting
lab.run_opt()

# Plot the results
lab.show_parameters
lab.plot()
```

This outputs the following:

```
The estimated parameters are:
k1 0.22638509313022112
k2 1.0031160083691573
```

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.

1.5.13 Using the Alternative Variance Estimation Method

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. 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.

```
    import kipet

    r1 = kipet.ReactionModel('reaction-1')

# Add the parameters
k1 = r1.parameter('k1', value=1.2, bounds=(0.01, 5.0))
    k2 = r1.parameter('k2', value=0.2, bounds=(0.001, 5.0))
```

(continues on next page)

(continued from previous page)

```
    # Declare the components and give the initial values
    A = r1.component('A', value=1.0e-3)
    B = r1.component('B', value=0.0)
    C = r1.component('C', value=0.0)

    # Define the reaction model
    r1.add_ode('A', -k1 * A )
    r1.add_ode('B', k1 * A - k2 * B )
    r1.add_ode('C', k2 * B )

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

    # Settings
    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.

```
r1.settings.variance_estimator.method = 'alternate'
r1.settings.variance_estimator.secant_point = 5e-4
r1.settings.variance_estimator.initial_sigmas = 5e-5
```

The new options include the method, which in this case is 'alternate', initial_sigmas, which is our initial value for the sigmas that we wish to start searching from, and the secant_point, which provides a second point for the secant method to start from. The final new option is the individual_species option. When this is set to False, we will obtain only the overall model variance, and not the specific species. Since the problem is unbounded when solving for this objective function, if you wish to obtain the individual species' variances, this can be set to True, however this should be used with caution as this is most likely not the real optimum, as the device variance that is used will not be the true value, as the objective functions are different.

```
# Perform parameter fitting
r1.run_opt()

# Display the results
r1.results.show_parameters
r1.plot()
```

Fig. 1.24: Concentration profiles for the alternate variance method

Fig. 1.25: Single species absorbance profiles for the alternate variance method

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.

1.5.14 Handling 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 an 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,

$$\mathbf{D} = \mathbf{C}\mathbf{S}^\top + \mathbf{G} + \mathbf{E}$$

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,

$$\mathbf{G} = \mathbf{e}\mathbf{g}^\top \tag{1.64}$$

$$\mathbf{e} = [1\ 1\ \dots\ 1]^\top_{(n_{tp}\times 1)} \tag{1.65}$$

$$\mathbf{g} = \begin{bmatrix} g_1\ g_2\ \dots\ g_{n_{wp}} \end{bmatrix}^\top \tag{1.66}$$

$$\mathbf{G} \in R^{n_{tp}\times n_{wp}} \tag{1.67}$$

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 _sub matrix (rko), which is composed of stoichiometric coefficient matrix St and dosing concentration matirx Z_in. (detailed derivation is omitted.) If rko > 0, G can be decomposed as,

$$\mathbf{G} = \mathbf{C}\mathbf{S}_g^\top \tag{1.68}$$

$$\mathbf{S}_g = \begin{bmatrix} \mathbf{s}_1^g\ \mathbf{s}_2^g\ \dots\ \mathbf{s}_{n_{wp}}^g \end{bmatrix} \tag{1.69}$$

$$\mathbf{S}_g \in R^{n_{wp}\times n_{wp}} \tag{1.70}$$

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

$$\mathbf{D} = \mathbf{C}\left(\mathbf{S} + \mathbf{S}_g\right)^\top + \mathbf{E}$$

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

If rko = 0, G cannot to decomposed. Therefore, the objective function of the parameter estimation problem should be modified as,

$$\sum_{i=1}^{n_{tp}}\sum_{l=1}^{n_{wp}}\left(d_{il} - \sum_{k=1}^{n_c} c_k(t_i)s_k(\lambda_l) - g(\lambda_l)\right)^2$$

For time variant unwanted contributions, G can be expressed as a rank-1 matrix as well,

$$\mathbf{G} = \mathbf{q}\mathbf{g}^\top \tag{1.71}$$

$$\mathbf{q} = \begin{bmatrix} q_1\ q_2\ \dots\ q_{n_{tp}} \end{bmatrix}^\top \tag{1.72}$$

$$\mathbf{g} = \begin{bmatrix} g_1\ g_2\ \dots\ g_{n_{wp}} \end{bmatrix}^\top \tag{1.73}$$

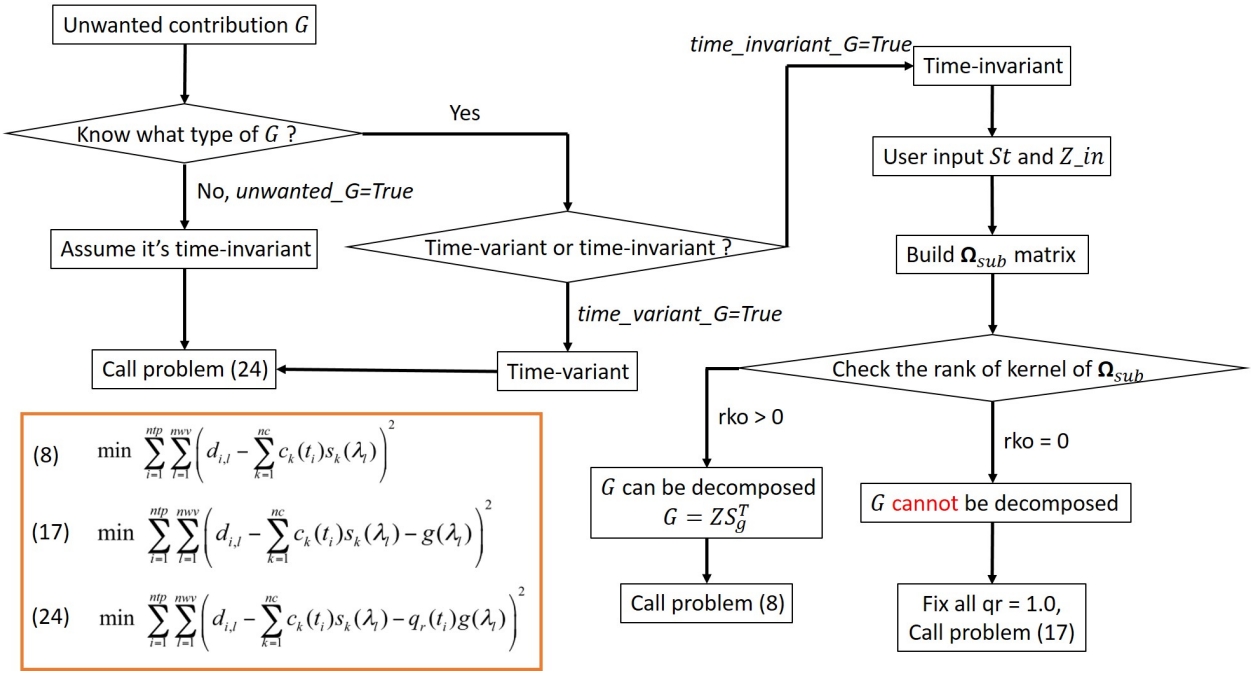
(1.74)

and the objective of problem is modified as follows,

$$\min \sum_{i=1}^{n_{tp}} \sum_{l=1}^{n_{wp}} \left(d_{il} - \sum_{k=1}^{n_c} c_k(t_i) s_k(\lambda_l) - q_r(t_i) g(\lambda_l) \right)^2$$

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 qr(i) and g(l), this will lead to nonunique values of these two variables and convergence difficulty in solving optimization problem. Therefore, we force qr(t_ntp) to be 1.0 under the assumption that qr(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,

```
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.
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.

```
r1.settings.parameter_estimator.G_contribution = 'time_invariant_G'
r1.settings.parameter_estimator.St = St
r1.settings.parameter_estimator.Z_in = Z_in
```

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.

```
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 ReactionModel 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,

$$\min \frac{(D - C_A S_A - C_B S_B)^2}{4 \times 10^{-8}} + \frac{(C_A - Z_A)^2}{1 \times 10^{-8}} + \frac{(C_B - Z_B)^2}{2 \times 10^{-8}}$$

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,

$$\min \frac{(D - C_A S_A - C_B S_B)^2}{1.0} + \frac{(C_A - Z_A)^2}{0.25} + \frac{(C_B - Z_B)^2}{0.5}$$

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.

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

1.5.15 Simultaneous Parameter Selection and Estimation Using the Reduced Hessian

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 estimabilty 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 initilizations. 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:

```
from pyomo.environ import exp
import kipet

r1 = kipet.ReactionModel('cstr')
r1.unit_base.time = 'hr'
r1.unit_base.volume = 'm**3'

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

# Add the model parameters
Tf = r1.parameter('Tf', value=293.15*factor, bounds=(250, 350), units='K')
Cfa = r1.parameter('Cfa', value=2500*factor, bounds=(100, 5000), units='mol/m**3')
rho = r1.parameter('rho', value=1025*factor, bounds=(800, 1100), units='kg/m**3')
delH = r1.parameter('delH', value=160*factor, bounds=(10, 400), units='kJ/mol')
ER = r1.parameter('ER', value=255*factor, bounds=(10, 500), units='K')
k = r1.parameter('k', value=2.5*factor, bounds=(0.1, 10), units='1/hour')
Tfc = r1.parameter('Tfc', value=283.15*factor, bounds=(250, 350), units='K')#,␣
↪fixed=True)
rhoc = r1.parameter('rhoc', value=1000*factor, bounds=(800, 2000), units='kg/m**3')#,␣
↪fixed=True)
h = r1.parameter('h', value=1000*factor, bounds=(10, 5000), units='W/m**2/K')#,␣
↪fixed=True)

# Declare the components and give the valueial values
A = r1.component('A', value=1000, variance=0.001, units='mol/m**3')
T = r1.state('T', value=293.15, variance=0.0625,  units='K')
Tc = r1.state('Tc', value=293.15, variance=0.001, units='K')

# Change this to a clearner method
full_data = kipet_model.read_data_file('data/all_data.csv')

F = r1.constant('F', value=0.1, units='m**3/hour')
Fc = r1.constant('Fc', value=0.15, units='m**3/hour')
Ca0 = r1.constant('Ca0', value=1000, units='mol/m**3')
V = r1.constant('V', value=0.2, units='m**3')
Vc = r1.constant('Vc', value=0.055, units='m**3')
Ar = r1.constant('Area', value=4.5, units='m**2')
Cpc = r1.constant('Cpc', value=1.2, units='kJ/kg/K')
```

(continues on next page)

(continued from previous page)

```
Cp = r1.constant('Cp', value=1.55, units='kJ/kg/K')

r1.add_data('T_data', data=full_data[['T']], time_scale='hour')
#r1.add_data('A_data', data=full_data[['A']].loc[[3.9, 2.6, 1.115505]], time_scale='hour')

# Not really necessary, but useful for tracking
rA = r1.add_reaction('rA', k*exp(-ER/T)*A, description='Reaction A' )

r1.add_ode('A', F/V*(Cfa - A) - rA )
r1.add_ode('T', F/V *(Tf - T) + delH/rho/Cp*rA - h*Ar/rho/Cp/V*(T -Tc) )
r1.add_ode('Tc', Fc/Vc *(Tfc - Tc) + h*Ar/rhoc/Cpc/Vc*(T -Tc) )

# Convert the units
r1.check_model_units(display=True)

r1.settings.collocation.ncp = 1
r1.settings.collocation.nfe = 150
```

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

```
rh_method = 'fixed'
results = r1.rhps_method(method='k_aug',
                         calc_method=rh_method)
```

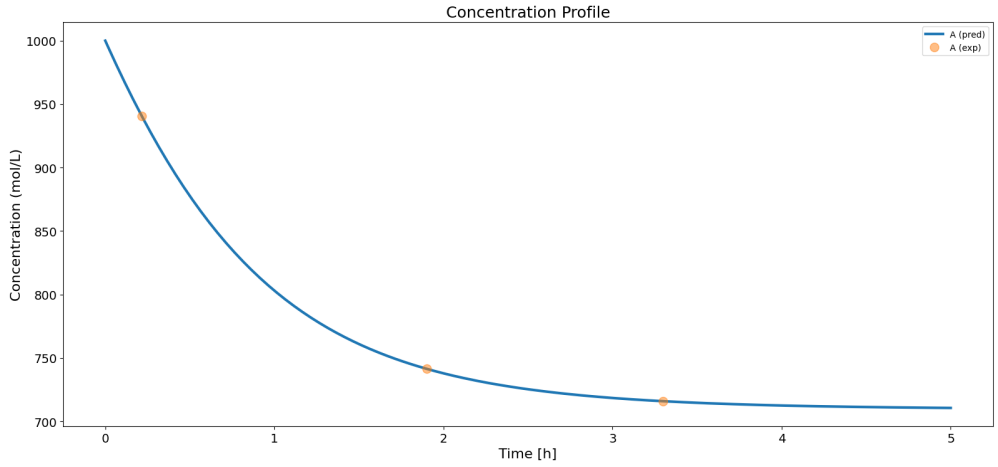


Fig. 1.26: Concentration profiles for the tutorial example 16. Noitce the addition of the three “experimental” points.

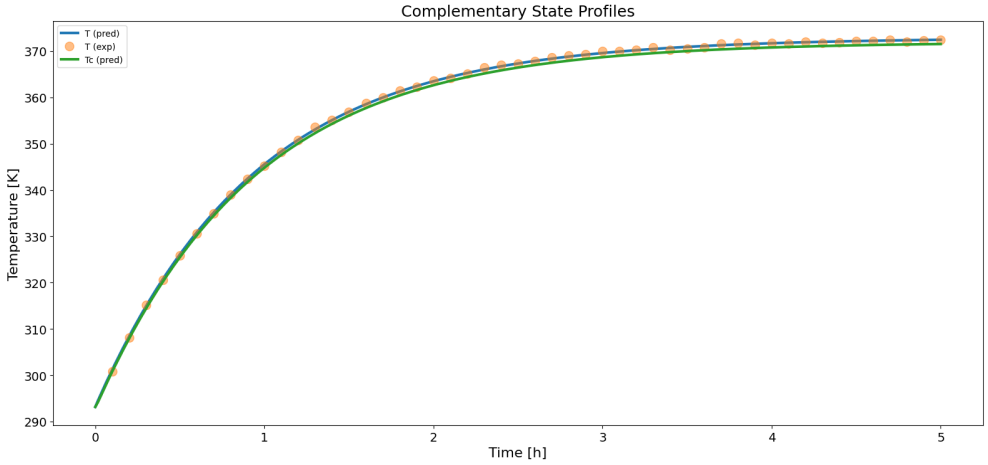


Fig. 1.27: Complementary state (here temperature) profiles for the tutorial example 15

1.5.16 Adding Custom Data for Parameter Fitting

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 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 only 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.

Note: If you use an expression with a division term that may go to zero, KIPET will automatically acount for this by adding very small amounts to both the numerator and denominator. In this way, the ratio will approach unity if both the numerator and the denominator approach zero.

```
import kipet

r1 = kipet.ReactionModel('reaction-1')
r1.unit_base.time = 's'

full_data = kipet.read_data('data/ratios.csv')

# Add the model parameters
k1 = r1.parameter('k1', value=2, bounds=(0.0, 10.0), units='1/s')
k2 = r1.parameter('k2', value=0.2, bounds=(0.0, 10.0), units='1/s')

# Declare the components and give the valueial values
A = r1.component('A', value=1.0, units='mol/l')
B = r1.component('B', value=0.0, units='mol/l')
```

(continues on next page)

(continued from previous page)

```
C = r1.component('C', value=0.0, units='mol/l')

r1.add_data('C_data', data=full_data[['A']], remove_negatives=True)
r1.add_data('y_data', data=full_data[['y']])

r1.add_ode('A', -k1 * A )
r1.add_ode('B', k1 * A - k2 * B )
r1.add_ode('C', k2 * B )

r1.add_expression('y', B/(B + C), description='Ratio of B to (B + C)' )

r1.check_model_units()

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

# Perform parameter fitting
r1.run_opt()

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

Fig. 1.28: Concentration profiles (notice that only A has been measured)

Fig. 1.29: Fitted custom data (ratio of B to B plus C)

1.5.17 Mixed Units in a Batch Reactor

Files Ex_20_fed_batch_step.py

In this example, we again look at the simple series reaction we have considered in many of the examples. The difference here is that we are modeling a batch reactor with a component feed that lasts only a portion of the reaction.

This example also shows how KIPET can automatically check the units provided and ensure that the resulting model uses consistent units.

Note: Since the ODEs are explicitly entered, the volume changes are not automatically added. In this example you can see that we did this manually, although this is not necessary.

```
import kipet

    r1 = kipet.ReactionModel('fed_batch')

    # Set the base time unit (match data)
r1.unit_base.time = 'min'
r1.unit_base.volume = 'L'
```

(continues on next page)

(continued from previous page)

```
# Reaction rate constant (parameter to fit)
k1 = r1.parameter('k1', value = 0.05, units='ft**3/mol/min')

# Components
A = r1.component('A', value=2.0, units='mol/L')
B = r1.component('B', value=0.0, units='mol/L')
C = r1.component('C', value=0.0, units='mol/L')

# Reactor volume
V = r1.volume(value = 0.264172, units='gal')

# Step function for B feed - steps can be added
s_Qin_B = r1.step('s_Qin_B', coeff=1, time=15, switch='off')

# Volumetric flow rate of the feed
Qin_B = r1.constant('Qin_B', value=6, units='L/hour')

# Concentration of B in feed
Cin_B = r1.constant('Cin_B', value=2.0, units='mol/L')

# Add the data
filename = 'data/abc_fedbatch.csv'
r1.add_data('C_data', file=filename, remove_negatives=True, time_scale='min')

# Convert your model components to a common base
# KIPET assumes that the provided data has the same units and will be
# converted as well - be careful!
#r1.check_component_units()

Qin = Qin_B * s_Qin_B

R1 = k1*A*B
QV = Qin/V

r1.add_ode('A', -A*QV - R1 )
r1.add_ode('B', (Cin_B - B)*QV - R1 )
r1.add_ode('C', -C*QV + R1)
r1.add_ode('V', Qin )

# Check for consistant units in the model equations
r1.check_model_units(display=True)

r1.run_opt()
r1.plot()
```

Fig. 1.30: Concentration profile results

Fig. 1.31: Volume change during the reaction

1.6 Citing KIPET

Please use the following when citing KIPET:

C. Schenk, M. Short, J.S. Rodriguez, D. Thierry, L.T. Biegler, S. García-Muñoz, W. Chen (2020) Introducing KIPET: A novel open-source software package for kinetic parameter estimation from experimental datasets including spectra, Computers & Chemical Engineering, 134, 106716. https://doi.org/10.1016/j.compchemeng.2019.106716

M. Short, L.T. Biegler, S. García-Muñoz, W. Chen (2020) Estimating variances and kinetic parameters from spectra across multiple datasets using KIPET, Chemometrics and Intelligent Laboratory Systems, https://doi.org/10.1016/j.chemolab.2020.104012

M. Short, C. Schenk, D. Thierry, J.S. Rodriguez, L.T. Biegler, S. García-Muñoz (2019) KIPET–An Open-Source Kinetic Parameter Estimation Toolkit, Computer Aided Chemical Engineering, 47, 299-304.

1.7 References

Chen, W., Biegler, L.T., 2020, Reduced Hessian based parameter selection and estimation with simultaneous collocation approach, AIChE Journal, 16242.

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

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

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

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

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

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

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

Schenk, C., Short, M., Rodriguez, J.S., Thierry, D.M., Biegler, L.T., García-Muñoz, S., Chen, W., 2020, Introducing KIPET: A novel open-source software package for kinetic parameter estimation from experimental datasets including spectra, Computers & Chemical Engineering, 134, 106716. https://doi.org/10.1016/j.compchemeng.2019.106716

Short, M., Schenk, C., Thierry, D.M., Rodriguez, J.S., Biegler, L.T., Garcia-Munoz, S., 2019, KIPET – An Open-Source Kinetic Parameter Estimation Toolkit, Computer-Aided Chemical Engineering, 47, 299-304

Short, M., Biegler, L.T., García-Muñoz, S., Chen, W., 2020, Estimating variances and kinetic parameters from spectra across multiple datasets using KIPET, Chemometrics and Intelligent Laboratory Systems, https://doi.org/10.1016/j.chemolab.2020.104012

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

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

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

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

1.8 Code Documentation

All of the public classes and methods used in KIPET are listed on these pages.

Users should generally only be interested in the modules and classes found in **Main modules**.

Developers may find the other modules and classes of interest.

1.8.1 Main modules

The two modules that will actually be used by the user are found here. They are initialized automatically and can be imported directly from "kipet".

main_modules

These are the two modules used directly by the user. Most, if not all, of the functionality of KIPET can be accessed here.

reaction_set

reaction_model

1.8.2 Calculation Tools

Several, more general, calculation methods are located here.

calculation_tools

This module contains various methods for performing critical calcultaions in KIPET.

beer_lambert

diagnostic_tools

Diagnostic Tools used in Kipet

Note: These methods should be checked for usefulness and possibly removed

kipet.calculation_tools.diagnostic_tools.**rank**(*A*, *eps=1e-10*)
obtains the rank of a matrix based on SVD

Parameters

- **A** (*numpy.ndarray*) – A numpy matrix

- **eps** (*float*) – the value of the singular values that corresponds to 0 when smaller than eps. Default = 1e-10

Return rank The rank of the matrix

Return type int

kipet.calculation_tools.diagnostic_tools.**nullspace**(*A*, *atol=1e-13*, *rtol=0*)

Obtains the nullspace of a matrix based on SVD. Taken from the SciPy cookbook

If *A* is an array with shape (m, k), then *ns* will be an array with shape (k, n), where n is the estimated dimension of the nullspace of *A*. The columns of *ns* are a basis for the nullspace; each element in numpy.dot(A, ns) will be approximately zero.

Parameters

- **A** (*numpy.ndarray*) – A numpy matrix
- **atol** (*float*) – The absolute tolerance for a zero singular value. Singular values smaller than *atol* are considered to be zero.
- **rtol** (*float*) – The relative tolerance. Singular values less than rtol*smax are considered to be zero, where smax is the largest singular value.

Return ns

Return type numpy.ndarray

kipet.calculation_tools.diagnostic_tools.**basic_pca**(*dataFrame*, *n=None*, *with_plots=False*)

Runs basic component analysis based on SVD

Parameters

- **dataFrame** (*pandas.DataFrame*) – spectral data
- **n** (*int*) – number of largest singular-values to plot
- **with_plots** (*bool*) – Argument for files with plots due to testing

Returns None

kipet.calculation_tools.diagnostic_tools.**perform_data_analysis**(*dataFrame*, *pseudo_equiv_matrix*, *rank_data*)

Runs the analysis by Chen, et al, 2018, based upon the pseudo-equivalency matrix. User provides the data and the pseudo-equivalency matrix and the analysis provides suggested number of absorbing components as well as whether there are likely to be unwanted spectral contributions.

Parameters

- **dataFrame** (*pandas.DataFrame*) – Spectral data
- **pseudo_equiv_matrix** (*list*) – List containing the rows of the pseudo-equivalency matrix.
- **rank_data** (*int*) – Rank of the data matrix, as determined from SVD (number of coloured species)
- **with_plots** (*bool*) – Argument for files with plots due to testing

Returns None

helper

Top Level helper classes for KIPET

class kipet.calculation_tools.helper.**AttrDict**(**args*, ***kwargs*)
This class lets you use nested dicts like accessing attributes using a dot notation

Methods

- *update()*

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

update(**args*, ***kwargs*)
Method used to update the AttrDict using the same syntax as the dict class

Parameters

- **args** (*tuple*) – The arguments (pass through)
- **kwargs** (*dict*) – The keyword arguments (pass through)

Returns None

class kipet.calculation_tools.helper.**DosingPoint**(*component*, *time*, *conc*, *vol*)
Small class to handle the dosing points in a cleaner manner

Parameters

- **component** (*str*) – The name of the model component
- **time** (*float*) – The time of the dosing
- **conc** (*tuple*) – A tuple of the concentration (conc (float), units (str))
- **vol** (*tuple*) – A tuple of the dosing volume (vol (float), units (str))

Methods

- *as_list()*

__init__(*component*, *time*, *conc*, *vol*)
Initialize self. See help(type(self)) for accurate signature.

property as_list
Return the dosing attributes as a list

Returns List of dosing point attributes

Return type list

interpolation

This module contains functions for ensuring that variables have congruent time indicies.

For many users who may have delved into older versions of KIPET, these were formerly found in the PyomoSimulator and Simulator classes.

kipet.calculation_tools.interpolation.**interpolate_trajectory**(*t*, *tr*)
Method for interpolating trajectories

Parameters

- **t** (*list*) – the list of times

- **tr** (*pandas.DataFrame*) – The trajectory data

Return float tr_val The interpolated value

kipet.calculation_tools.interpolation.**interpolate_trajectory2**(*t*, *data*)
Takes some data and fills in the missing points using interpolation

Parameters

- **t** (*float*) – The time point
- **data** (*pandas.DataFrame*) – The data to interpolate with

Return df_interpolated The data after interpolation

Return type pandas.DataFrame

prob_gen_tools

Problem Generation tools for Kipet

kipet.calculation_tools.prob_gen_tools.**gaussian_single_peak**(*wl*, *alpha*, *beta*, *gamma*)
Helper function to generate absorption data based on Lorentzian parameters

Parameters

- **wl** (*float*) – Wavelength
- **beta** (*float*) – Beta
- **alpha** (*float*) – Alpha
- **gamma** (*float*) – gamma

Returns The single peak

Return type float

kipet.calculation_tools.prob_gen_tools.**absorbance**(*wl*, *alphas*, *betas*, *gammas*)
Helper function to generate absorption data based on Lorentzian parameters

Parameters

- **wl** (*float*) – Wavelength
- **betas** (*array-like*) – Beta
- **alphas** (*array-like*) – Alpha
- **gammas** (*array-like*) – gamma

Returns The sum of single peaks

Return type float

kipet.calculation_tools.prob_gen_tools.**generate_absorbance_data**(*wl_span*, *parameters_dict*)
Helper function to generate absorption data based on Lorentzian parameters

Parameters

- **wl_span** (*array-like*) – Array of wavelengths
- **parameters_dict** (*dict*) – The dictionary of alphas, betas, and gammas

Returns data_frame

Return type pandas.DataFrame

kipet.calculation_tools.prob_gen_tools.**generate_random_absorbance_data**(*wl_span*, *component_peaks*, *component_widths=None*, *seed=None*)

Helper function to generate absorption data based on Lorentzian parameters

Parameters

- **wl_span** (*array-like*) – Array of wavelengths
- **component_peaks** (*dict*) – Dictionary with number of component peaks
- **component_widths** (*dict*) – Optional widths for components (otherwise 1000)
- **seed** (*int*) – Random seed number

Returns data_frame

Return type pandas.DataFrame

kipet.calculation_tools.prob_gen_tools.**add_noise_to_signal**(*signal*, *size*)

Adds a random normally distributed noise to a clean signal. Used mostly in Kipet To noise absorbances or concentration profiles obtained from simulations. All values that are negative after the noise is added are set to zero

Parameters

- **signal** (*pandas.DataFrame*) – The Z or S matrix to have noise added to it
- **size** (*float*) – sigma (or size of distribution)

Returns: :return: Noised data :rtype: pandas.DataFrame

reduced_hessian

1.8.3 Estimability Tools

Tools for analyzing parameters are found here.

estimability_tools

Methods used to check parameter estimability.

estimability_analysis

parameter_handling

parameter_ranking

reduced_hessian_parameter_selection

1.8.4 Estimator Tools

The main estimation modules and classes are found here alongside several helper modules. The core methods for variance estimation and parameter fitting (VarianceEstimator and ParameterEstimator classes) are found here.

estimator_tools

These include the primary methods and classes for variance and parameter estimation.

additional_inputs

Additional Input Options for Variance and Parameter Estimation Classes

kipet.estimator_tools.additional_inputs.**add_inputs**(*est_object*, *kwds*)

This method takes an estimator object and some keyword arguments. The keyword arguments are then placed into the correct variables or methods.

All changes are made in place. This used to be inside the estimator classes, but is cleaner if left here.

Parameters

- **est_object** – This is a VarianceEstimator or ParameterEstimator instance
- **kwds** (*dict*) – A dict of keyword arguments

Returns None

estimator_methods

fe_factory

fe_simulator

jumps_method

multiple_experiments

parameter_estimator

pyomo_simulator

results_object

variance_estimator

1.8.5 Settings

General settings for KIPET are stored here.

general_settings

These include several modules dealing with settings and other options.

settings

Settings for KIPET

class kipet.general_settings.settings.**Settings**(*category='model'*)

This is a container for all of the options that can be used in Kipet Since it can be confusing due to the large number of options, this should make it easier for the user to see everything in one spot.

This class loads the default settings from the settings.yml file located in the directory wherever KIPET has been installed. If you really want to change this file, go ahead, but make sure that you save a copy of the original settings in case something goes wrong.

Methods

- `updated_settings()`

__init__(*category='model'*)

Settings object initialization that begins each time a ReactionModel instance is created.

update_settings(*category*, *item*, *value*)

Sets the default settings to some new value

This allows the user to make permanent changes to the settings file.

> **Warning:** Careful! This may result in KIPET not working properly if you make a change that is incompatible!

Parameters

- **category** (*str*) – The category containing the value to change
- **item** (*str*) – The name of the setting to change
- **value** (*str*) – The new value for the setting

Returns None

property as_dicts

unit_base

Unit Base - Object for holding units and the unit registry

class kipet.general_settings.unit_base.**UnitBase**

Class to hold universal units and the unit registry

Parameters

- **ur** (*pint.UnitRegistry*) – default unit registry
- **time** (*str*) – time base
- **volume** (*str*) – volume base

- **concentration** (*str*) – concentration base

__init__()
Initialize the UnitBase for the ReactionModel

Parameters

- **ur** (*pint.UnitRegistry*) – default unit registry
- **time** (*str*) – time base
- **volume** (*str*) – volume base
- **concentration** (*str*) – concentration base

variable_names

General KIPET Model Varialbe names

class kipet.general_settings.variable_names.**VariableNames**
Names of the KIPET model variables

This provides a central location to reduce errors in developing KIPET modules by forcing a naming convention. It also provides useful collections of variables for use in many other functions.

This is not really needed by the user and should only be used by developers.

__init__()
Initialize self. See help(type(self)) for accurate signature.

property optimization_variables
These are the independent variables that need to be fixed in simulations

property time_dependent_variables
Property to return variables that are time dependent:

Defaults ['Z', 'dZdt', 'S', 'C', 'X', 'dXdt', 'U', 'Y']

Return model_vars The list of target variables

Return type list

property modeled_states
Property to return variables that are modeled:

Defaults ['Z', 'dZdt', 'X', 'dXdt']

Return model_vars The list of target variables

Return type list

property model_vars
Property to return component variables:

Defaults ['Z', 'X', 'P', 'Y', 'step', 'Const']

Return model_vars The list of target variables

Return type list

property rate_vars
Property to return rate variables:

Return model_vars The list of target variables

Return type list

property plot_vars
Property to return plotted variables

Return model_vars The list of target variables

Return type list

1.8.6 Input/Output

Input and output methods specific to KIPET.

input_output

These include the primary methods and classes for reading and writing data.

kipet_io

read_hessian

read_write_tools

1.8.7 Mixins

Mixin classes used in other parts of KIPET

mixins

Mixins used in other KIPET classes are maintained here.

parameter_estimator_mixins

wavelength_mixins

1.8.8 Model Components

Methods and classes used in building the models used in KIPET

model_components

The model components used in KIPET are maintained here.

component_expression

data_component

element_blocks

expression

model_components

objectives

This file contains the object functions used throughout Kipet modules in one place. This should reduce the redundancy in using objective terms

Note: These methods are not complete in all cases and may not be used in future versions of KIPET.

kipet.model_components.objectives.**get_objective**(*model*, **args*, ***kwargs*)
Main method to gather the objective terms

Parameters

- **model** (*ConcreteModel*) – The model for which the objective will be built
- **args** (*tuple*) – The arguments to be passed
- **kwargs** (*dict*) – The arguments to be passed

Returns An objective component for the model

Return type Objective

kipet.model_components.objectives.**conc_objective**(*model*, **args*, ***kwargs*)
Method to build concentration terms in the objective function

Parameters

- **ConcreteModel model** (*Pyomo*) – This is the current model used in parameter fitting
- **args** (*tuple*) – Arguments to be passed
- **kwargs** (*dict*) – Arguments to be passed

Returns obj

Return type expression

kipet.model_components.objectives.**comp_objective**(*model*, **args*, ***kwargs*)
Method to build individual complementary state terms in the objective function

Parameters

- **ConcreteModel model** (*Pyomo*) – This is the current model used in parameter fitting
- **args** (*tuple*) – Arguments to be passed
- **kwargs** (*dict*) – Arguments to be passed

Returns obj

Return type expression

kipet.model_components.objectives.**spectra_objective**(*model*, **args*, ***kwargs*)
Method to build individual spectral terms in the objective function

Parameters

- **ConcreteModel model** (*Pyomo*) – This is the current model used in parameter fitting
- **args** (*tuple*) – Arguments to be passed
- **kwargs** (*dict*) – Arguments to be passed

Returns obj

Return type expression

kipet.model_components.objectives.**absorption_objective**(*model*, **args*, ***kwargs*)
Method to build individual absorption terms in the objective function

Parameters

- **ConcreteModel model** (*Pyomo*) – This is the current model used in parameter fitting
- **args** (*tuple*) – Arguments to be passed
- **kwargs** (*dict*) – Arguments to be passed

Returns obj

Return type expression

spectral_handler

Spectral Data Handling for Kipet

class kipet.model_components.spectral_handler.**SpectralData**(*name*, *data=None*, *file=None*, *remove_negatives=False*)
This class is used to handle the spectral data used in a ReactionModel

Since spectral data is different from the state data and requires different methods to modify the data, a separate class was designed to house all of the spectra specific methods.

Parameters

- **name** (*str*) – The name for the data set
- **data** (*pandas.DataFrame*) – The spectral data (D matrix)
- **remove_negatives** (*bool*) – Option to set negative values to zero

Methods

- *add_data()*
- *reset()*
- *plot()*
- *savitzky_golay()*
- *snv()*
- *msc()*
- *baseline_shift()*
- *decrease_wavelengths()*

- *decrease_times()*

__init__(*name*, *data=None*, *file=None*, *remove_negatives=False*)
Initialize a SpectralData instance

Parameters

- **name** (*str*) – The name for the data set
- **data** (*pandas.DataFrame*) – The spectral data (D matrix)
- **remove_negatives** (*bool*) – Option to set negative values to zero

add_data(*data*)
Adds a dataset to a SpectralData instance.

This is used only if the SpectralData instance is created without a data attribute (not being None). This handles setting up the data_orig attribute as well and is therefore better than simply using setattr.

Parameters **data** (*pandas.DataFrame*) – The spectral data (D matrix)

Returns None

reset()
Resets the data back to the originally supplied data

Returns None

plot(*data_set='data'*)
Plots spectral data in 3D plot.

Plots the modified or original data sets.

Parameters **data_set** (*pandas.DataFrame*) – attribute name of the spectral data

Returns None

savitzky_golay(*window=3*, *poly=2*, *deriv=0*, *in_place=True*)
Implementation of the Savitzky-Golay filter for Kipet. Used for smoothing data, with the option to also differentiate the data. Can be used to remove high-frequency noise. Creates a least-squares fit of data within each time window with a high order polynomial centered centered at the middle of the window of points.

Parameters

- **window** (*int*) – The length of the window. Must be an odd integer number
- **poly** (*int*) – Order of the polynoial used in the filter. Should be less than window_size-1
- **deriv** (*int*) – (optional) The order of the derivative to compute (default = 0 means only smoothing)
- **in_place** (*bool*) – Option to update the data in place

Returns DataFrame containing the smoothed data

Return type pandas.DataFrame

References This code is an amalgamation of those developed in the scipy.org cookbook and that employed in Matlab by WeiFeng Chen. Original paper: A. Savitzky, M. J. E. Golay, Smoothing and Differentiation of Data by Simplified Least Squares Procedures. Analytical Chemistry, 1964, 36 (8), pp 1627-1639.

snv(*offset=0*, *in_place=True*)
Implementation of the Standard Normal Variate (SNV) filter for Kipet which is a weighted normalization method that is commonly used to remove scatter effects in spectroscopic data, this pre-processing step can

be applied before the SG filter or used on its own. SNV can be sensitive to noisy entries in the spectra and can increase nonlinear behaviour between S and C as it is not a linear transformation.

Parameters

- **offset** (*float*) – User-defined offset which can be used to avoid over-normalization for samples with near-zero standard deviation. Guide for choosing this value is for something near the expected noise level to be specified. Default value is zero.
- **in_place** (*bool*) – Option to update the data in place

Returns DataFrame containing pre-processed data

Return type pandas.DataFrame

msc(*reference_spectra=None*, *in_place=True*)

Implementation of the Multiplicative Scatter Correction (MSC) filter for Kipet which is simple pre-processing method that attempts to remove scaling effects and offset effects in spectroscopic data. This pre-processing step can be applied before the SG filter or used on its own. This approach requires a reference spectrum which must be determined beforehand. In this implementation, the default reference spectrum is the average spectrum of the dataset provided, however an optional argument exists for user-defined reference spectra to be provided.

Parameters

- **reference_spectra** (*pandas.DataFrame*) – Optional user-provided reference spectra argument. Default is to automatically determine this using the average spectra values
- **in_place** (*bool*) – Option to update the data in place

Returns DataFrame pre-processed data

Return type pandas.DataFrame

baseline_shift(*shift=None*, *in_place=True*)

Implementation of basic baseline shift. 2 modes are avaliable: 1. Automatic mode that requires no user arguments. The method identifies the lowest value (NOTE THAT THIS ONLY WORKS IF LOWEST VALUE IS NEGATIVE) and shifts the spectra up until this value is at zero. 2. Baseline shift provided by user. User provides the number that is added to every wavelength value in the full spectral dataset.

Parameters

- **shift** (*float*) – user-defined baseline shift
- **in_place** (*bool*) – Option to update the data in place

Returns DataFrame containing pre-processed data

Return type pandas.DataFrame

decrease_wavelengths(*A_set=2*, *specific_subset=None*, *in_place=True*)

Takes in the original, full dataset and removes specific wavelengths, or only keeps every multiple of A_set. Returns a new, smaller dataset that should be easier to solve

Parameters

- **A_set** (*int*) – optional user-provided multiple of wavelengths to keep. i.e. if 3, every third value is kept. Default is 2.
- **specific_subset** (*array-like*) – (optional) If the user already knows which wavelengths they would like to remove, then a list containing these can be included.

param bool in_place: Option to update the data in place

Returns DataFrame with the smaller dataset

Return type

decrease_times(*A_set=2*, *in_place=True*)

Takes in the original, full dataset and removes specific wavelengths, or only keeps every multiple of A_set. Returns a new, smaller dataset that should be easier to solve

Parameters **A_set** (*array-like*) – optional user-provided multiple of wavelengths to keep. i.e. if 3, every third value is kept. Default is 2.

param bool in_place: Option to update the data in place

Returns DataFrame with the smaller dataset

Return type pandas.DataFrame

units_handler

1.8.9 Model Tools

Methods and classes used in performing various tasks on the models used in KIPET

model_tools

diagnostics

model_functions

pyomo_model_tools

replacement

scaling

template_builder

visitor_classes

1.8.10 Variance Methods

Core methods used to estimate the variances taken from the literature.

variance_methods

alternate_method

chen_method

chen_method_pyomo

chen_method_scipy

1.8.11 Visuals

Plotting tools

visuals

plots

reports

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THREE

KIPET RESOURCES

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 modeling language and much of the syntax can be found here:

https://pyomo.readthedocs.io/en/stable/

PYTHON MODULE INDEX

k

INDEX

Symbols

A

B

C

D

G

I

K

M

N

O

P

R

S

T

U

V