1 Data Assimilation and State Estimation in SOFA

Optimus plugin was created to provide a testing environment for data-driven physics-based modeling (typically finite elements, FE). While actually the plugin implements only stochastic methods based on Kalman filtering, its architecture allows for implementation of generic prediction–correction schemes where the model is employed as a predictor and correction is performed using given observation data.

In this text, a model $\mathcal{M}$ performs a transformation of state $\mathbf{s}$ given parameters $\mathbf{p}$ and control $\mathbf{c}$, i. e.,

$$\mathbf{s} \xrightarrow{\mathcal{M}(\mathbf{p},\mathbf{c})} \mathbf{s}'. \tag{1}$$

Typically, the model is represented by SOFA components such as force fields. The state is given by positions and optionally velocities, and control is provided as applied forces, pressures and/or prescribed displacements. Parameters are given by the type of model; typically, these are mechanical parameters (Young's modulus, spring stiffness).

It is assumed that one or combination of following types of estimation is performed:

- **State estimation:** typical scenario in Kalman filtering where the state $\mathbf{s}$ of the model is assumed to suffer from uncertainties due to the initial conditions (i. e., the state at time t_0) and/or modeling itself. The correction aims at improving the estimation of the state given the observations.
- **Parameter identification:** this scenario is usually known as *data assimilation*: the main source of uncertainty is related to the parameters of the model, i. e., uncertanty related to the stiffness of an organ. The goal is thus to *learn* the actual value of the parameter(s) given the observations.
- **Joint state–parameter estimation:** while the parameters are source of uncertainty, even a perfect knowledge of the parameters does not provide completely reliable results since the model itself (even if accurately parametrized) is imperfect, and/or there is an inherent uncertainty related to the initial conditions.

Recently, in the context of estimation of boundary conditions, we have been interested in **control estimation**, for example, estimation of applied forces. To summarize, Optimus plugin allows for estimation or identification of any quantity that can be *delivered* to a SOFA component. either through a data engine (parameters, forces) or via write accessors (positions, velocities).

1.1 State handling and animation loop

Adopting the naming conventions in both the mechanics and filtering, we end up with two notions of states:

Mechanical state: Configuration of the object being simulated, typically given by positions and velocities. This state is always deterministic, i. e. it is represented by a single vector of mechanical degrees of freedom (DoFs). This state is typed using SOFA types (Vec3d, Rigid3d and similar). In this text, mechanical state is abbreviated as *M-state*.

Stochastic state: Quantity given as a probability distribution (PD), represented by its statistics. In case of Gaussian PD considered here, the statistics are the expected value and the standard deviation. From the physical point of view, this state may contain an arbitrary combination of positions, velocities, parameters and forces. It is stored a set of Eigen vectors (e.g., two of them for Gaussian PD). In this text, stochastic state is abbreviated as *S-state*.

In Optimus, *M-state* is strictly separated from *S-state*both conceptually and from the implementation point of view and these two terms cannot be interchanged. While the filter works **only** with *S-state*, classical SOFA components (e.g. components from SOFA framework and other existing plugins) employ only *M-state*. The interface between *S-state* and *M-state*is provided by the `StochasticStateWrapper` and `OptimParams` components.

2 Data Assimilation description

2.1 Parameter handling

The main goal of the stochastic state data assimilation (*SSDA*) (and data assimilation in general) is to estimate a set of model parameters based on the model dynamics and observations, where both are associated with uncertainties.

From the SOFA point of view, the parameters are usually simple numbers (or vectors of numbers) handled by local components: for example, the Young's modulus is a number (usually constant) defined and handled by the FEM-component, stiffnesses of a set of springs is a vector handled by the spring component.

However, in estimation process, the parameters are included in the extended state (*S-state* in our case) and they evolve together with the model state. The parameters are not regarded as simple numbers, but as probability distributions (defined by corresponding statistics as mean value and variance). Since the model itself requires concrete instances the parameters, these are instantiated in each `Forward` step using the method of *$\sigma-$point*.

In order to integrate the two different concepts, a new component was introduced, namely `OptimParams`:

- The component is a container of parameters which vary during the *SSDA*. The parameters are defined using *initial value* and *standard deviation*.
- It also handles correct update of the *S-state*: it works as an *interface* between SOFA and Filtering components.
- If option `optimize` is true (which is the case by default), **the node containing the component is considered as the part of the physical model assimilated by filtering process**. If in a node this component is not present, or `optimize="0"`, the object modeled by the node is not included in the assimilation.

In each phase and step of the *SSDA*, the actual value of the parameters handled by `OptimParams` can be accessed via SOFA `Data<> value`. Therefore, the components which depends on the parameters being optimized (such as already mentioned Young's modulus) must be linked via @. **It must be verified, that the component which depends on the parameters being optimized is compatible with the fact that this parameter changes in each step,** i. e. there are no internal structures which are precomputed using the initial values of the parameters.

The `OptimParams` component was conceived to be as general as possible, therefore it can handle structures defined as `Data<DataTypes>` where `DataTypes` is the parameter of template (which could be virtually any type (scalars, vectors of scalars, *VecCoord* etc.). In order to achieve this degree of generality, the concept of *functors* was employed as can be seen in the code of the component.

Finally, in one node, several `OptimParams` can be present if different types of parameters are to be optimized (this has been test only partially. . .).

2.2 Prediction: `Forward`

In the forward stage, the filter performs the prediction phase by evaluating on *operator* (UKF) and/or it's derivatives (EKF). This evaluation is done for each $\sigma-point$ independently and the result is used to compute the mean prediction (*a priori*) value of the *S-state*. The *operator* $\mathcal{O}$ is in fact a time step of the model, here represented by a standard SOFA simulation time step. A few facts about $\mathcal{O}$ must be emphasized:

- From the filtering context, it acts on the state (*S-state*), so for a given *S-state* X_n it performs $X^-_{n+1} = \mathcal{O}(X_n)$.
- From the SOFA point of view, it performs a single step of the simulation on a given DoF positions/velocities (encoded as a part of X_n) while taking into account the current parameters (also encoded as a part of X_n).
- It is evaluated several times in each step of *SSDA*, namely, it is evaluated for each $\sigma-point$ which is a different perturbation of X_{n+1} according to the chosen method (simplex, star, circle).

In order to facilitate the integration with SOFA, it was desirable to make the real evaluation of $\mathcal{O}$ (SOFA model) completely transparent from the filtering process. For this reason, a new component `StochasticStateWrapper` was created: it serves as a wrapper of all physical simulation components and ensures the correspondence between the respective mechanical states and the *S-state*.

This part is also the trickiest one in terms of the implementation as the SOFA components are usually not implemented with the idea of total independence on the software.

The forward stage is computationally expensive mainly due to multiple evaluation of the model; the subsequent manipulation with *S-state* are rather cheap. At the same time, the execution of the operator is embarrassingly parallel. However, the main issue is that for a parallelization on N threads, N independent copies of the SOFA world would have to be created (TODO, with python maybe?).

2.3 Correction: `Analyze`

While the forward stage requires a significant manipulation of *S-state* and simulation components, analyze stage is almost exclusively performed by filter. The only part supplied by SOFA is the `ObservationManager` which computes an *innovation* using prediction computed previously and observations obtained by an external source.

So far, we have been experimenting only with simple cases where the observations are generated using a *forward simulation*, i. e. a standard SOFA simulation where the states of the modeled objects are stored in each step into a text file using SOFA monitors. Then, each simulation scene has to have an *observation sub-node* which contains following components:

- Mechanical object which defines a set of observation points. These can be any points inside the moving objects were observations are made (trivially, it can be equivalent to the mechanical state of the assimilated object).
- Mapping between the observation mechanical state and the mechanical state of the assimilated body (trivially an `IdentityMapping`, usually a `BarycentricMapping`).
- An observation manager which provides a method `GetInnovation` called by the filter. The method must have an access to observation in each time (supplied for example by an `ObservationSource` and it uses mapping (`Apply`) to get the observations in points defined by the mechanical state.

So far we are using `SimulationStateObservationSource` in order to read the positions exported by monitors in forward simulations. Currently, the code allows for adding a noise, although only in very preliminary version.

3 Components

`FilteringAnimationLoop`

- Compliant with SOFA API for animation loops.
- Calls filter wrapper to perform the assimilation in each step.
- Issues special events: PredictionEndEvent after calling the prediction step of the filter, and CorrectionEndEvent after calling the correction step of the filter.

`StochasticStateWrapper`

- Wrapper of model provided by SOFA. This wrapper implements the interface between the stochastic components (filters) and SOFA simulation.
- Must be placed inside the SOFA subnode containing the physical simulation components.
- Requires MechanicalObject and assumes OptimParams in the same node.

`PreStochasticWrapper`

- Special version of wrapper which allows for including other simulated object into Optimus scene which do not contain any quantities being estimated.
- Typical example is an obstacle which displays a physical behaviour, however, none of its features is directly involved in the estimation.

`UKFilter`

- Unscented Kalman Filter proposed by Julier and Uhlman (1997). Implementation performed according to code in Reduced order Kalman filter implemented according to Moireau, Philippe, and Dominique Chapelle. "Reduced-order Unscented Kalman Filtering with application to parameter identification in large-dimensional systems."
- Main methods are initializeStep(), computePrediction() and computeCorrection(), all called by the `FilteringAnimationLoop`.

`ROUKFilter`

- Reduced order Kalman filter implemented according to Moireau, Philippe, and Dominique Chapelle. "Reduced-order Unscented Kalman Filtering with application to parameter identification in large-dimensional systems."
- Main methods are initializeStep(), computePrediction() and computeCorrection(), all called by the `FilteringAnimationLoop`.

`UKFilterSimCorr`

- Special version of UKF filter purely for data assimilation. Currently being studied.
- Main methods are initializeStep(), computePrediction() and computeCorrection(), all called by the `FilteringAnimationLoop`.

`MappedStateObservationManager`

- One instance of observation manager which handles the observations and provides them to the filter via computation of innovation.
- Assumes that the predicted observations are mapped to the main mechanical object (associated with `StochasticStateWrapper`).
- Templated on `DataTypes1` and `DataTypes2` since it calls mapping apply (having also two template parameters).
- Inherits from `ObservationManager` which is only an abstract base.
- The uncertainty of the observations is set using option `observationStdev` which is the standard deviation of observations assumed in the data assimilation.
- Noise can be added using `noiseStdev`: the noise is generated using `Boost`, has a normal distribution with zero mean and standard deviation given by `noiseStdev`.

`OptimParams`

- Container of Gaussian stochastic variables given by expected value and standard deviation.
- Templated on any numerical type in SOFA, specializations needed.
- Abstract pure base `OptimParamsBase` created to facilitate the manipulation and define the API.
- In *forward mode*, it can be used either as a simple container of parameters which remain constant, or as a *parameter controller* which can change the parameters during the simulation. The change is defined using option `prescribedParamKeys="`T^0 v_0^0 `...` v_n^0 T^1 v_0^1 `...` v_n^1 `... ...` T^t v_0^t `...` v_n^t`"` where the values of parameters are defined in each time T^t (not a time step, but the simulation time). Here, the option `numParams` is required in order to parse the keys correctly. The transition between two different values can be done smoothly (based on `tanh`), the option `interpolateSmooth`. Finally, the initial value in the XML file is ignored and only the keys are used so that the real value of the parameter is interpolated for given time step.
- The value of parameteres in each time step can be exported using option `exportParamFile`.
- In the *filtering mode*, only the initial value `initValue` together with the standard deviation `stdev` are set in the XML file.
- In some optimization cases, it's desirable to keep some properties of the parameters being optimized (e.g positivity for the Young's modulus. The option `transformParams` can be used to choose between: 0 the no transformation is done, 1: absolute value of the parameters is taken, before feeding the model. 2: sigmoid transformation of the parameters is taken, before feeding the model. 3: exponential transformation of the parameters is taken, before feeding the model. 4: projection of the parameters to the bounded space is taken, before feeding the model. The bounds of the projected space are defined using `minValue` and `maxValue` fields.

`SimulatedStateObservationSource`

- Auxiliary components that reads and provides observations stored in a monitor file generated by a forward simulation.
- Mainly supplies `GetObservation` method called by the observation manager.
- Requires correct prefix of the monitor-generated file.

4 Data

The plugin contains several volume (VTK) and surface (STL) meshes used in the examples, benchmarks and scenes. Most of meshes were generated by GMesh generator.

5 Visualization

Optimus plugin has several python scripts that allow to show the dynamics of estimated parameters (stiffnesses), variance values, and correlation between parameters. All scripts use Matplotlib python library to draw figures and charts and YAML configuration files to load main scene parameters.

All visualization scripts are stored in python_src/visualisation subfolder in Optimus. To scripts generally require only path to output folder(s) with results, since special subfolders structure, which is recognized by scripts, is generated and YAML is copied to output folder during data assimilation process.

6 Examples

This is a description of an old examples, which are now replaced by anothers.

6.1 EX1: Estimation of constant Young's moduli of segments of a heterogeneous object under gravity

The data are generated using `CylinderModulusForward.scn` and in each step, the entire mechanical object of the cylinder is saved by the `Monitor` component. The cylinder is divided into three parts and each one is assigned different Young's modulus using OptimParams (which are here used exclusively as a container).

Data assimilation is performed by `CylinderModulusROUKF.scn` so that:

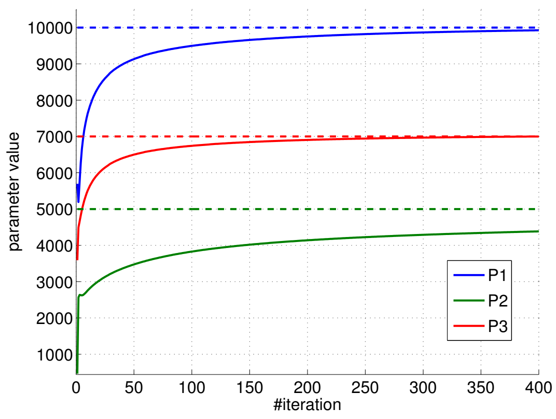
- The observations are read from the file generated by `Monitor` in the forward simulation.
- Either all DoFs can be used directly as observations (then `BarycentricMapping` is replaced by the `IdentityMapping`), or a subset of points inside the cylinder are chosen as the observation points mapped to the cylinder via `BarycentricMapping`.
- The variance of error is set via `observationStdev` which is the standard deviation of the observation error which is the measure of the uncertainty in observations. Note: zero cannot be used as the co-variance matrix cannot be inverted in this case.

- A white Gaussian noise can be added to the observations via the option `noiseStdev`. In general, `observationStdev` should be equal to `noiseStdev`: we use two separated options for experimental purposes.
- The initial values of the estimated parameters together with initial standard deviation is set in `OptimParams`.
- The vector of the actual estimations of the parameters can be stored at the end of each step using a non-empty filename `paramFileName` and the variance matrix is stored in a vectorial form using a non-empty filename `paramVarFileName`: first the diagonal, then off-diagonal elements under and above the diagonal, respectively (which is redundant as the matrix is symmetric).

Visualization:

- Cylinder: different colors associated to different Young's moduli.
- Red points: the actual position of the observation points on the model with estimated parameters
- Blue points: the actual position of the observation points loaded from the `Monitor` file with added noise.

Several examples of the assimilation are given below for different settings in the data-assimilation scene, showing the evolution of the parameter estimation and the variance (standard deviation squared) of the parameters.



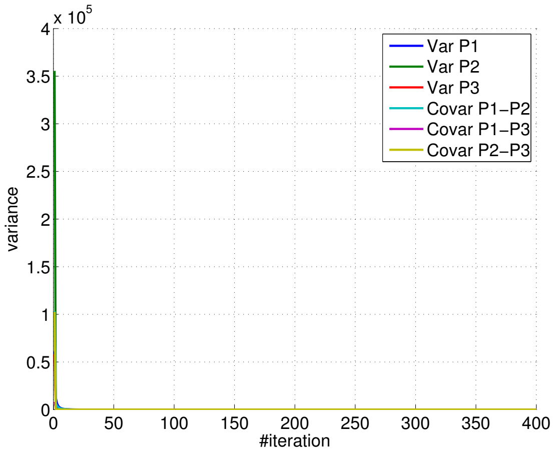


Figure 1: Parameters: `initValue="6000"`, `stdev="2000"`. Observations: all DoFs ($3\times$number of mesh nodes), `observationStdev=1e-4` no noise added. A naive scenario, where everything is known and no uncertainty exists.

6.2 EX2: Estimation of varying Young's moduli of segements of a heterogeneous object under gravity

The example is the same as the previous one, except for the fact that the Young's modulus is decreased from 6 kPa to 2 kPa between t=3 s and t=5 s. The result is illustrated in figures (dash line is showing the value of parameters).

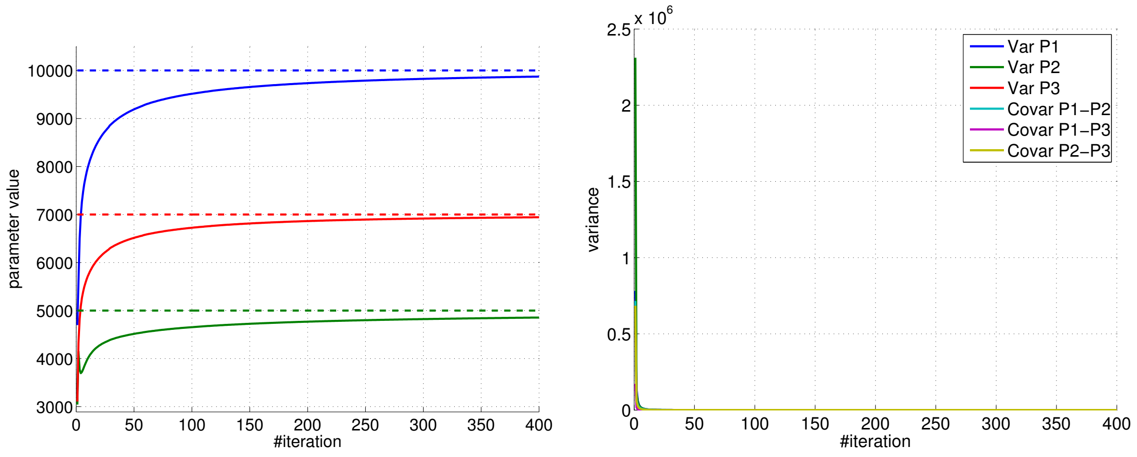


Figure 2: Parameters: `initValue="6000"`, `stdev="2000"`. Observations: 10 points selected along the centerline, `observationStdev=1e-4` no noise added. Still quite naive scenario, where number of observations is limited, but still no uncertainty exists.

7 Notes

7.1 Static vs. Dynamic

The graphs in the EX1 were obtained using quasi-static scenario, i. e. `StaticSolver` having `applyIncrementFactor="1"`. Observations:

- The result with the `StaticSolver` depend on `dt`: the results were obtained with `dt=0.01s`. Comparable results can be achieved with `dt=0.1`, the *SSDA* remains stable even for dt of 1 s.
- The stability of the *SSDA* seems to depend on the integration scheme, as with `EulerImplicitSolver` the method seems to be less stable. The stability depends on `Rayleigh stiffness`: a behavior comparable to the static behavior is achieved with RS=0.5, the process is stable and the final approximation is even more precise that in the static case.
- With decreasing RS, the DA becomes less stable in the beginning since the change in positions is too fast and so the perturbations becomes “wild”.

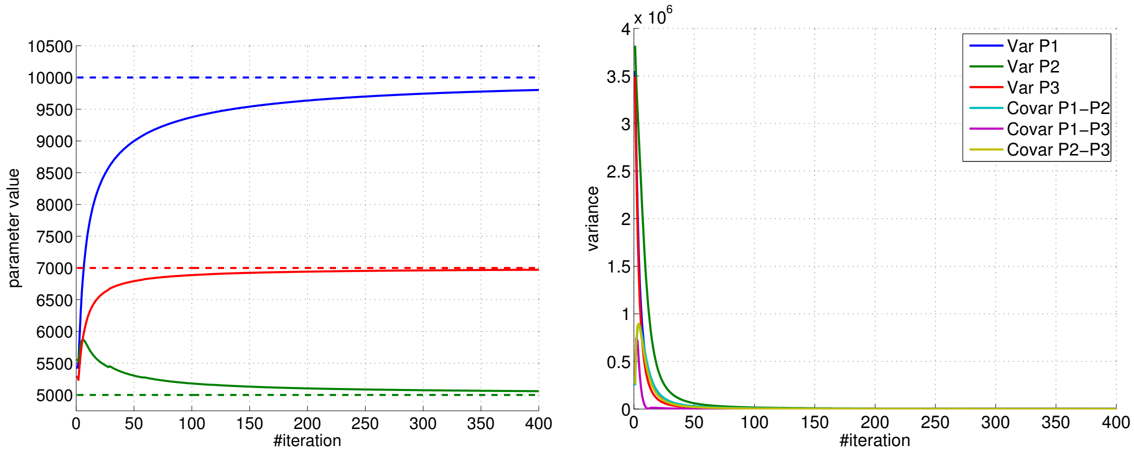


Figure 3: Parameters: `initValue="6000"`, `stdev="2000"`. Observations: 10 points selected along the centerline, `observationStdev=1e-3` no noise added. Although no noise added, we assume uncertainty of 1 mm for each observation.

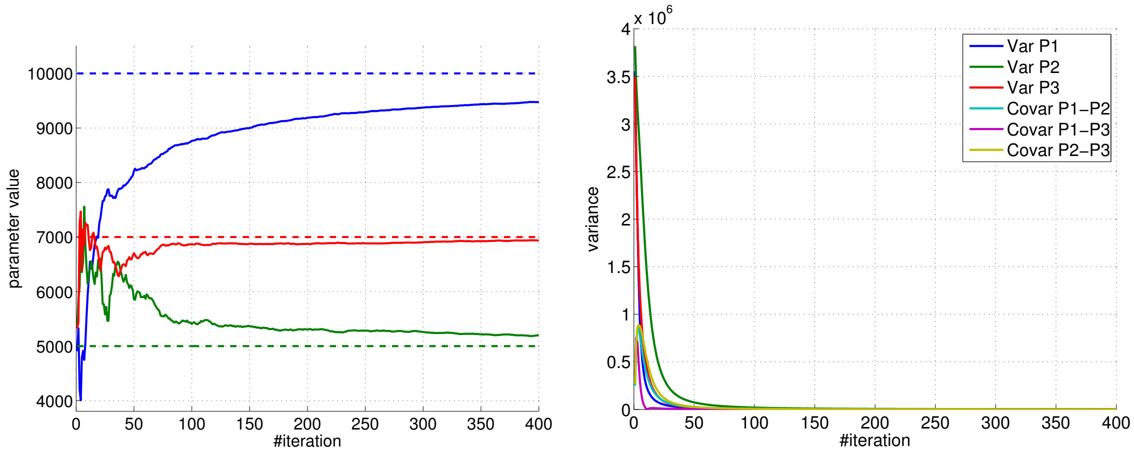


Figure 4: Parameters: `initValue="6000"`, `stdev="2000"`. Observations: 10 points selected along the centerline, `observationStdev=1e-3`, `noiseStdev=1e-3`. We have a real uncertainty of 1 mm modeled with white Gaussian noise with standard deviation of 1 mm.

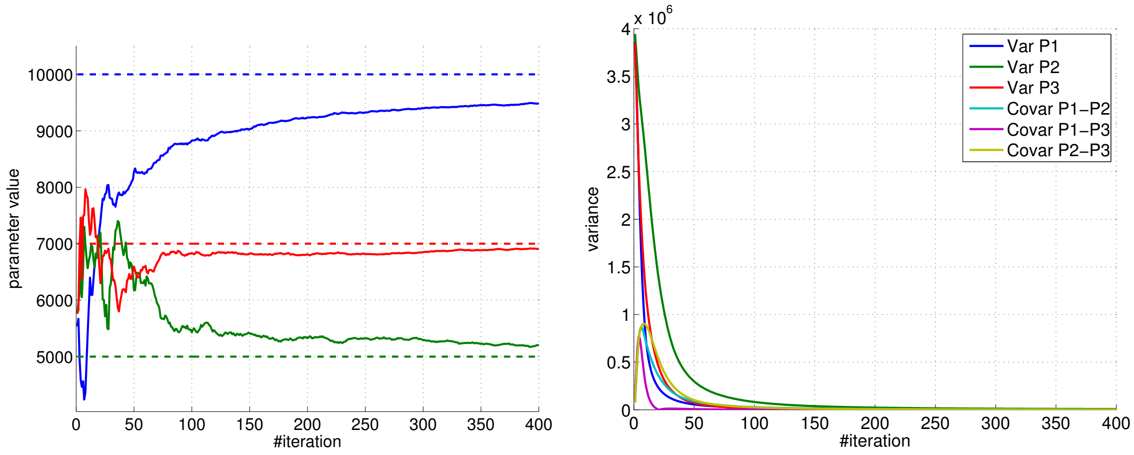


Figure 5: Parameters: `initValue="6000"`, `stdev="2000"`. Observations: 10 points selected along the centerline, `observationStdev=2e-3`, `noiseStdev=2e-3`. We have a real uncertainty of 2 mm modeled with white Gaussian noise with standard deviation of 2 mm.

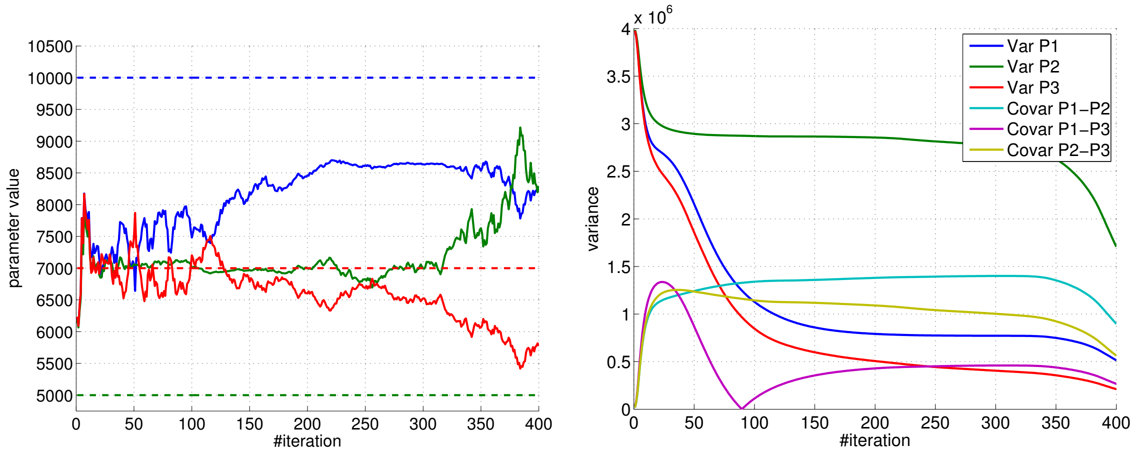


Figure 6: Parameters: `initValue="6000"`, `stdev="2000"`. Observations: 1 point selected in the middle of the cylinder, `observationStdev=2e-3`, `noiseStdev=2e-3`. An extreme case with fairly limited number of observations (a single point) and a real uncertainty of 2 mm modeled with white Gaussian noise with standard deviation of 2 mm.

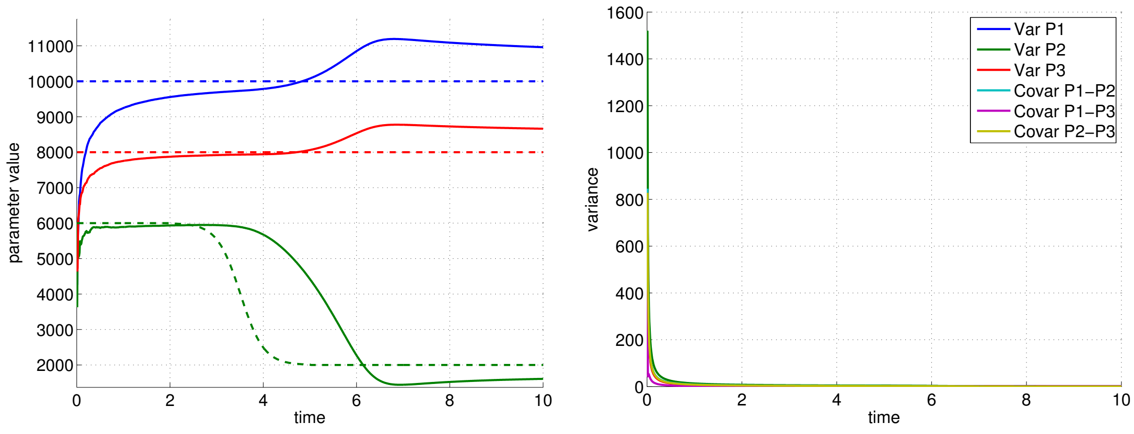


Figure 7: Parameters: `initValue="6000"`, `stdev="2000"`. Observations: 10 points selected along the centerline, `observationStdev=1e-4`, `noiseStdev=1e-4`. Even with no noise and very high confidence in measurement, the filter reacts to the change of the parameter with delay. Surprisingly, the other two parameters are perturbed as well.

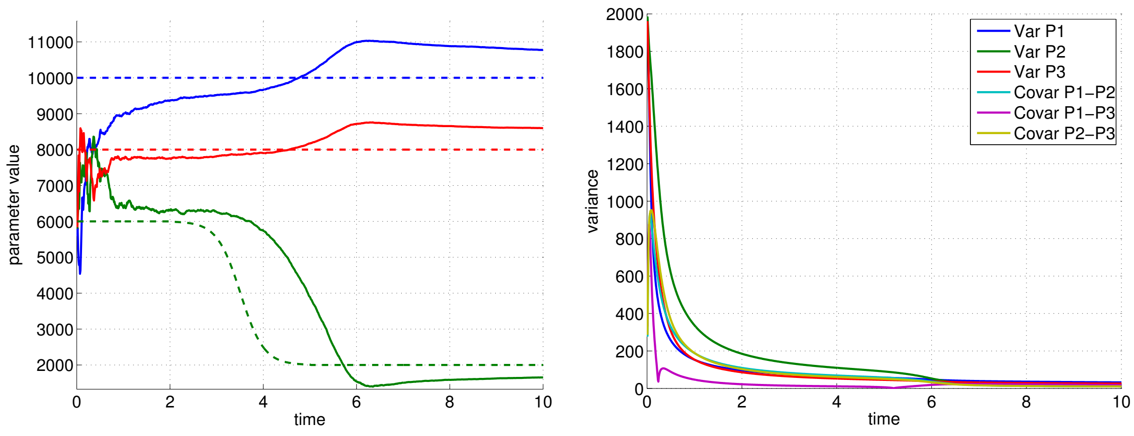


Figure 8: Parameters: `initValue="6000"`, `stdev="2000"`. Observations: 10 points selected along the centerline, `observationStdev=2e-3`, `noiseStdev=2e-3`. Despite the noise, the behavior of the filter is quite similar to the previous case when speaking about the adaptation of the varying parameter.

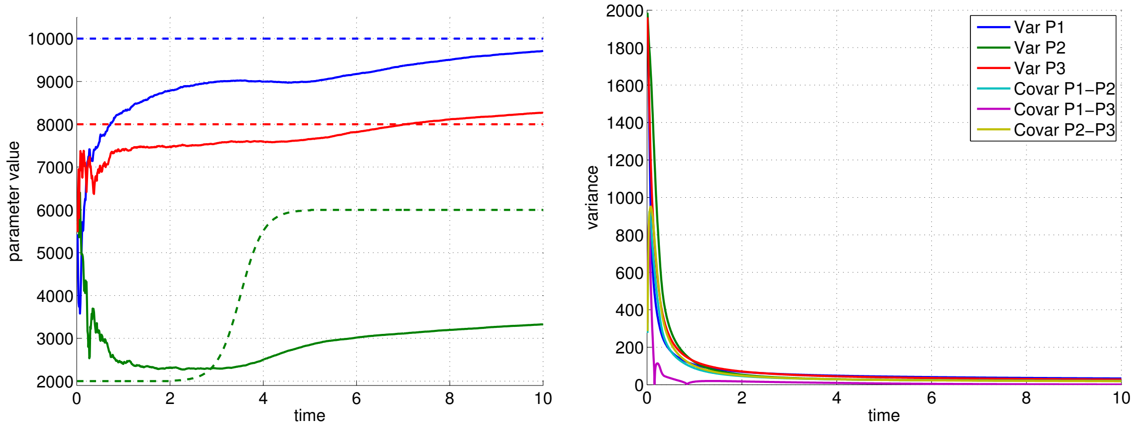


Figure 9: Parameters: `initValue="6000"`, `stdev="2000"`. Observations: 10 points selected along the centerline, `observationStdev=2e-3`, `noiseStdev=2e-3`, hardening instead of softening. varying parameter.