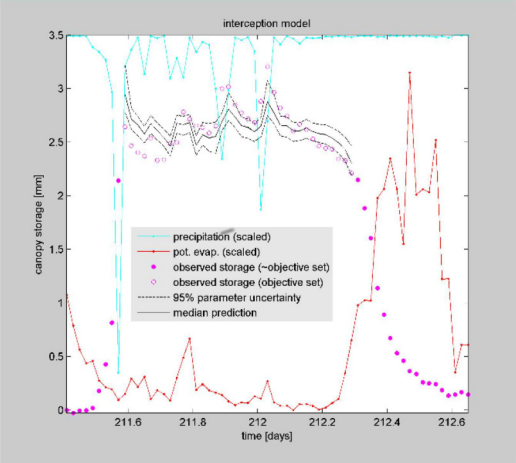
netherlands eScience center

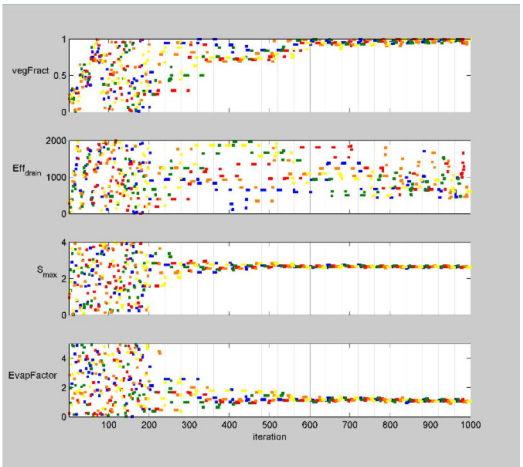
UNIVERSITEIT VAN AMSTERDAM

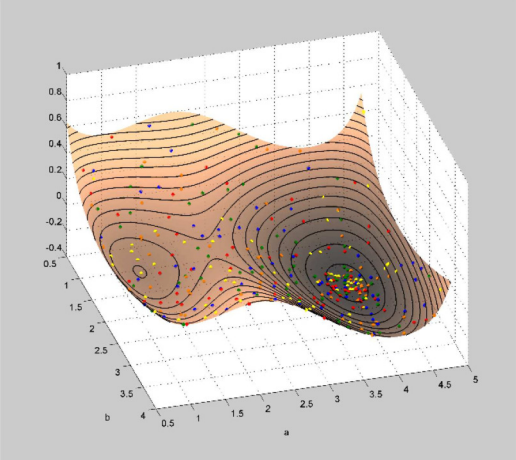
Inverse modeling

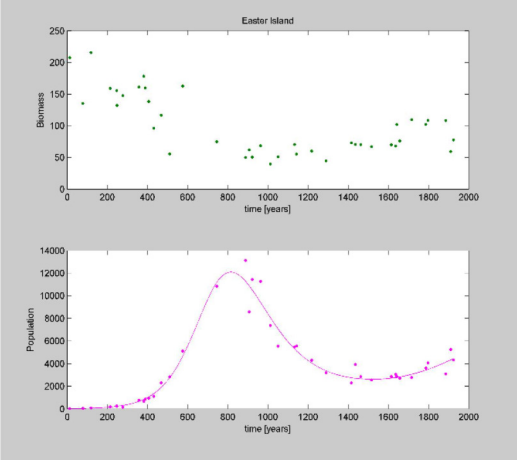
for improving Hydrological, Environmental and Ecological models

Authors: Willem Bouten, Jurriaan H. Spaaks, Bart Sweerts, Lotte de Vries
Date: March 2017









Contents

Chapter 1

Introduction

Dear participant, welcome to the course on Inverse Modeling. The document you have before you contains all of the exercises you will be making as part of the course. There are two types of exercise, labeled differently:

▶ 1. The exercises indicated by a black solid triangle are the core exercises; most of these will be discussed in class. Any files needed for completing the exercises can be found in the appropriate subdirectory of './exercises/'. As you can see, the exercises are numbered for convenient referencing.

▷ 2. The exercises indicated by an open triangle are optional exercises; these, you can make if the topic has your specific interest or if you feel you could use some more training, or just for fun!

Feel free to ask any of us if something isn't clear. You are also encouraged to discuss your results with the other participants. We hope you'll enjoy the course!

Chapter 2

Getting started

Aims:

- to understand differences between commonly used local search methods;
- to be aware of the assumptions in the linear approximation of parameter uncertainty:
- to experience the limitations of local search methods for complex response surfaces;
- to be aware of the existence of multiple minima in complex response surfaces;
- to acknowledge the added value of global search methods;
- to understand the need for efficient search methodologies.

2.1 Slug injection: manual calibration

A classic method in hydrology for determining the transmissivity and storage coefficient of an aquifer is called the slug test. A known volume of water Q [m^3] (the slug) is injected into a well, and the resulting effect on the head h [m] (i.e. water table elevation) at an observation well a distance d [m] away from the injection is monitored at times t [hr]. The measured head typically increases rapidly and then decreases more slowly. We wish to determine the storage coefficient S [$\mathsf{m}^3 \cdot \mathsf{m}^{-3}$] (a measure of the ability of the aquifer to store water) and the transmissivity T [$\mathsf{m}^2 \cdot \mathsf{hr}^{-1}$] (a measure of the ability of the aquifer to conduct water). The mathematical model for the slug test is:

$$h = \frac{Q}{4 \cdot \pi \cdot T \cdot t} e^{-d^2 \cdot S/(4 \cdot T \cdot t)} \tag{2.1}$$

▶ 1. Use the MATLAB editor to open 'mancal_sluginj.m' from './exercises/manual-calibration-slug-injection/'. Run the script by pressing F5. A Graphical User Interface will appear. Use this interface to calibrate parameters S and T of Equation 2.1. Assume $Q = 50$ [m^3] and $d = 60$ [m].

▶ 2. How do S and T affect the simulated pressure head?

Since this is a 2-parameter problem, we can easily visualize the objective function. Run 'respsurf.m'. The objective function is the sum of squared residuals (SSR).

► 3. Were you able to identify the "optimal" model parameters using manual calibration?

► 4. Looking at the response surface (Figure 2.1), do you think that the optimal model parameters are correlated?

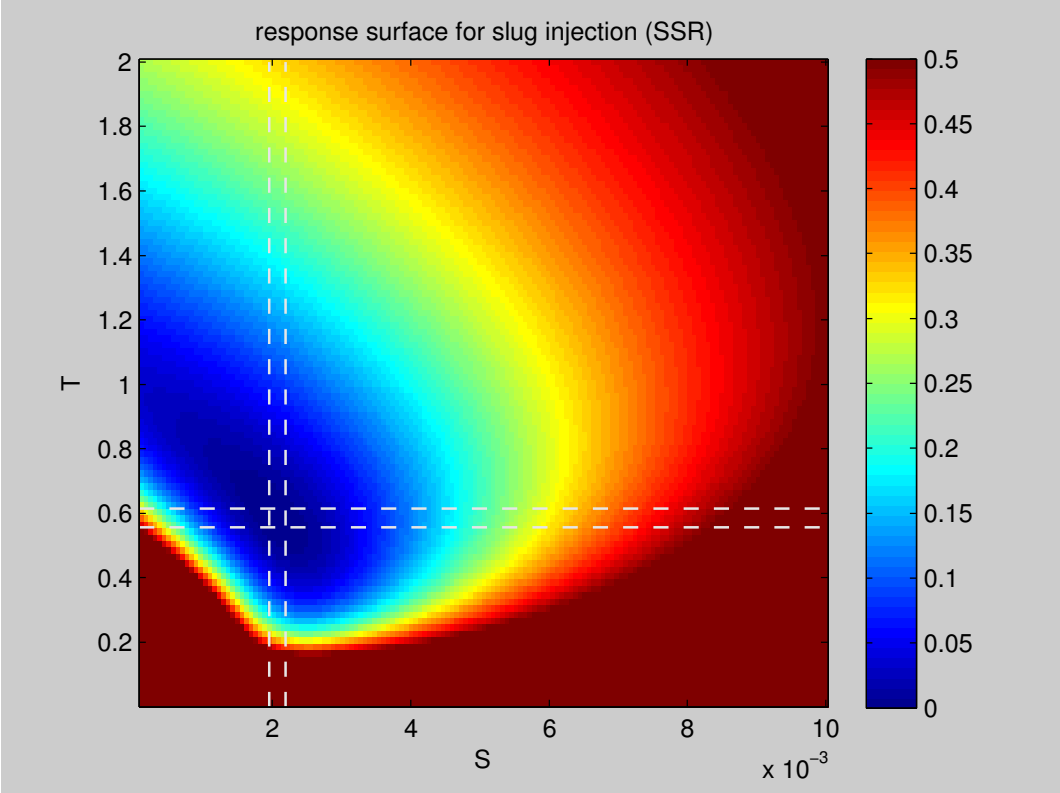


Figure 2.1: Response surface for the slug injection test. Dotted lines represent the 95% confidence interval of the model parameters when the standard deviation of the head measurement is 0.01 [m].

► 5. The dotted lines in Figure 2.1 indicate the 95% confidence intervals of the model parameters when the standard deviation of the head measurement is 0.01 [m]. Do you think this is a reasonable approximation for this nonlinear problem?

2.2 Optimization using local methods

2.2.1 Gauss-Newton and Levenberg-Marquardt

► 6. Two algorithms—Gauss-Newton (GN) and Levenberg-Marquardt (LM)–have been implemented to automatically find the optimal parameters for the slug test. To run these algorithms for the slug test problem, set your work directory to './exercises/local-methods-sluginj'. This directory contains a function that performs a Gauss-Newton as well as a

Levenberg-Marquardt optimization. This function requires an input argument containing a starting point (e.g. `[0.15,0.4]`). For each method, the function returns an array with a record of parameter sets and their objective score. You can call the function from the command line using:

```
[P_GN,P_LM]=Run_Optimization([0.15,0.4]);
```

▶ 7. Have a look at the code and interpret the output.

▶ 8. How quickly does the GN method converge?

▶ 9. How does this compare to the LM method?

▶ 10. Play around with different starting points for the two algorithms. Does the GN method always converge to the optimum? What about the convergence of the LM method?

▶ 11. Find a starting point where the GN and LM methods differ and study the iterations in detail (`P_GN` and `P_LM`).

▶ 12. What is the key difference between the iterations of the GN and LM method?

2.2.2 Multi-start simplex

HYMOD (Boyle et al., 2000) is a simple rainfall-runoff model with 5 model parameters (see Figure 2.2).

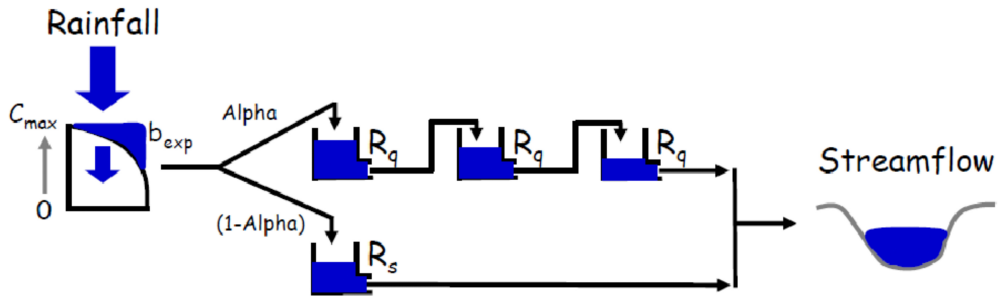


Figure 2.2: Structure of the HYMOD rainfall-runoff model.

with:

C_{max}	Maximum storage of the watershed
b_{exp}	Spatial variability of soil moisture capacity
$Alpha$	Partitioning factor of excess rainfall into slow and quick flow
R_s	Residence time of the slow tank
R_q	Residence time of the quick tanks

In this section, we will use a multi-start simplex (Nelder and Mead, 1965) to optimize the parameters of the HYMOD model. Specifically, we investigate whether the objective function (SSR) of this model has local minima that may complicate the use of local optimization methods such as Gauss-Newton, or Levenberg-Marquardt. To do this, 20 simplex runs are started from randomly determined starting points, which are iterated until convergence is achieved.

▶ 13. First, have a look at the code of 'Multistart_Simplex.m' in './exercises/local-methods-hymod/'. As you can see, the code uses the built-in MATLAB function `fminsearch`, which is an implementation of the simplex search algorithm. Run `Multistart_Simplex` (it'll run for a few minutes). Store the output by copy-pasting screenshots into a PowerPoint presentation.

▶ 14. Do all the Simplex runs converge to the same point in the parameter space?

▶ 15. What does this tell you about the presence of local minima?

▶ 16. For the slug injection test, we visualized the response surface. Can we make a similar figure here?

▶ 17. When you look at the results of the simulations with optimized parameters, why do you see just 3 or 4 time series and not 20?

▶ 18. How do you value the results of the various models?

▷ 19. In the previous exercise, a relatively short data set (3 months) was used to calibrate HYMOD. Extend the calibration period to 12 months by changing the values of `Extra.calPeriod` in 'Multistart_Simplex.m' and run the optimization again.

▷ 20. For this new subset of data, why did the number of local minima increase or decrease?

▷ 21. What is the quality of the model runs corresponding to the local minima?

▷ 22. If we discard the poor models, can you give an interpretation why these local minima occurred?

2.3 Global methods: Differential Evolution

Differential Evolution (Storn and Price, 1997) is a basic global optimization algorithm that uses a population of points in the parameter space to find the global maximum (minimum, if appropriate) of an objective function. Generally speaking, the population's properties are used to generate more points in those parts of the parameter space which are most promising with regard to yielding the global optimum.

▶ 23. Use the MATLAB editor to open './exercises/differential-evolution/respsurf.m'. This script contains a few lines of code to help you get started on your algorithm. Run the script to visualize the response surface for the benchmark function '6' (Shifted Rosenbrock's Function) for `x = [65:0.1:80]` and `y = [35:0.1:45]` (Figure 2.3).

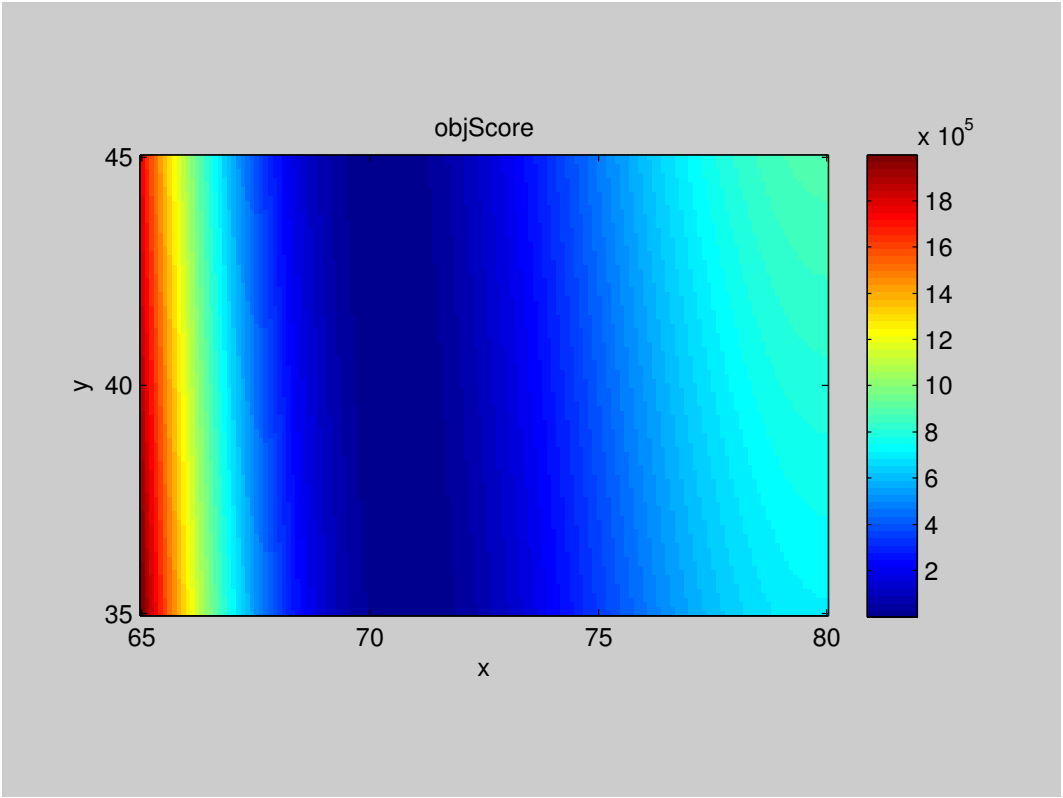


Figure 2.3: Response surface for the shifted Rosenbrock function.

The `benchmark_func` function offers not just 1, but 25 functions that can be visualized in the same way as you already did for the Rosenbrock function. You can choose different functions by changing the `funcFlag`. Each function has its own limits on the parameter space (see Table 2.1).

► 24. Experiment with different functions to get some idea of what their response surfaces look like.

In the next few exercises, you will write your differential evolution algorithm. We will take you through the following basic steps (see also Fig. 2.4 and Code Snippets 2.1 and 2.2):

1. generate the initial sample;
2. assign the initial sample to a new array `parents`;
3. for each sample in `parents`, calculate the corresponding objective score;
4. use `parents` to calculate new `proposals`;
5. for each sample in `proposals`, calculate the corresponding objective score;
6. accept either a sample from `proposals` or the corresponding sample from `parents` as the child, thus making an array `children` of the same size as `parents`;
7. assign `children` to `parents` for the next generation, and go back to step 4.

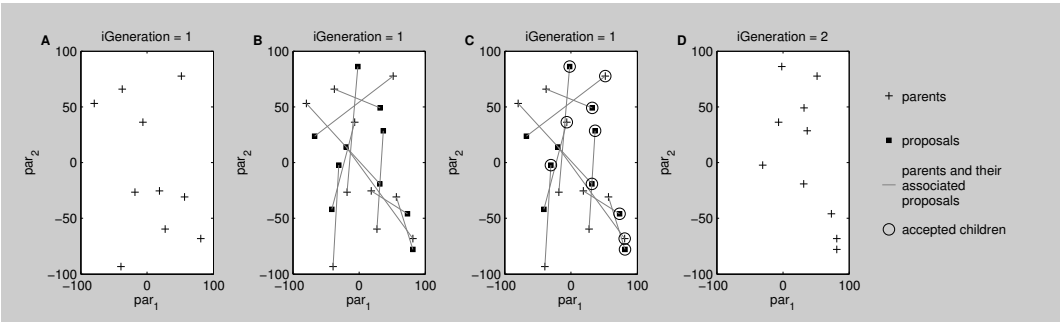


Figure 2.4: Differential evolution in a nutshell, using a population of `nPop = 10` to avoid cluttering the figures. **A**: Generation of the initial sample. **B**: Generation of proposal points. **C**: Accepting either a sample from `parents` or from `proposals` as a row in `children`. **D**: The `children` from one generation are the `parents` for the next generation.

► 25. As a first step, save your `respsurf` script under a new name 'runDiffEvo.m'.

► 26. Extend 'runDiffEvo.m' by creating an array `parents`, which has `nPop=50` rows, each of which represents a uniform random sample from the parameter space (see also Code Snippet 2.1). `parents` has `nDims+1` columns, i.e. the number of dimensions for which you want to solve `benchmark_func` plus one column to store the objective score. Refer to Table 2.1 for the parameter space limits. For now, stick with `nDims=2` and `funcFlag=6`, but you can change that later.

Code Snippet 2.1: Generating samples drawn from a uniform distribution spanning the parameter space.

```
% define the number of samples in the population
nPop = 50;

% define the number of dimensions ( =length of the parameter vector)
nDims = 2;

% set the lower limits for every dimension of the parameter vector, repeat
% for the number of members in the population
parSpaceLowerBounds = repmat([-100, -100], [nPop,1]);

% set the upper limits for every dimension of the parameter vector, repeat
% for the number of members in the population
parSpaceUpperBounds = repmat([100, 100], [nPop, 1]);

% calculate the range between the lower and upper boundaries on the
% parameter space for each dimension
parSpaceRange = parSpaceUpperBounds - parSpaceLowerBounds;

% perform nPop * nDims random draws from a uniform distribution
uniformRandomDraw = rand(nPop,nDims);

% fill the parents array with uniform draws within the parameter space
parents(1:nPop,1:nDims) = parSpaceLowerBounds + uniformRandomDraw .* parSpaceRange;
```

▶ 27. Calculate the last column of `parents` by running the objective function for each row.

▶ 28. Generate the `proposals` array (same size as `parents`). To do this, select the first sample from `parents`, as well as three other samples (`r1`, `r2`, `r3`), chosen at random from `parents` (MATLAB's built-in function `randperm` can be useful for this; see Code Snippet 2.2). The proposal is calculated as the position of the parent + F*`dist1` + K*`dist2`, in which `dist1` is the distance between the parent and `r1`, `dist2` is the distance between `r2` and `r3`, and F and K are settings that are part of the Differential Evolution algorithm. In the literature, F=0.6 and K=0.4 are common. Repeat for all samples in `parents`.

▶ 29. For each member in `proposals`, calculate the objective score.

▶ 30. Determine whether to choose the i^{th} sample of `parents` or the i^{th} sample of `proposals`, depending on which one has a better objective score. Assign to the i^{th} row in a new array `children`.

▶ 31. Assign `children` to `parents` for the next generation.

▶ 32. Wrap most of the above steps in a `for` loop, such that your script will repeatedly generate new (and hopefully better) children for 250 generations.

▶ 33. It's a little bit difficult to see if your script is doing what it is supposed to be doing, so you need to do some visualization. Copy and paste from 'vis-helper.txt' to create a figure similar to Figure 2.5.

▶ 34. During the optimization, you may notice that there are some points that are really persistent, even though they are well away from the global optimum. Explain why these points are so persistent.

▶ 35. The histogram in Figure 2.5 is bimodal. How does that relate to the answer to the previous question?

Code Snippet 2.2: Generating proposal points given a parent population.

```
% define which columns in parents and in proposals hold the parameter vector
parCols = 1:nDims;

% preallocate the space needed by the samples and their corresponding
% objective score
proposals = repmat(NaN, [nPop, nDims + 1]);

% iterate over the members of the population
for iPop = 1:nPop

    % draw 4 integer numbers from 1 to nPop
    v = randperm(nPop, 4);

    % if iPop happens to be drawn, remove it
    v(v == iPop) = [];

    % retrieve selected rows from the parents array; use only the parameter
    % vector columns
    r1 = parents(v(1), parCols);
    r2 = parents(v(2), parCols);
    r3 = parents(v(3), parCols);

    % calculate the two distances dist1 and dist2
    dist1 = r1 - parents(iPop, parCols);
    dist2 = r3 - r2;

    % calculate the proposal point
    proposals(iPop, parCols) = parents(iPop, parCols) + F * dist1 + K * dist2;
end
```

2.4 Differential Evolution with Metropolis

Over the next few exercises, you will add Metropolis acceptance to your Differential Evolution optimization, such that it can be used to quantify parameter uncertainty.

▷ 36. Clean up 'runDiffEvo.m' and save it as 'runDiffEvoMetro.m'. You will be making your edits in the latter script.

Scroll down to where 'runDiffEvoMetro.m' accepts either a member from `parents` or a member from `proposals` as a member of `children`. Currently, the proposal will only be accepted when it is better than the parent. In contrast, the Metropolis acceptance scheme is crucially different in that it will *sometimes* accept a proposal parameter combination that is actually *worse* than the parent.

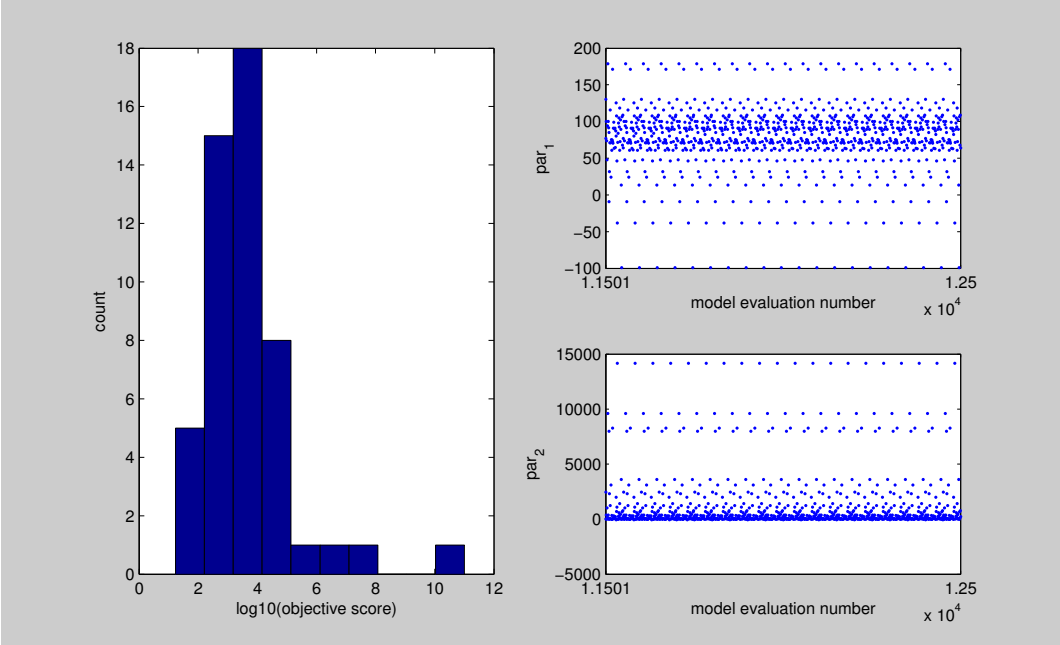


Figure 2.5: Visualization results for the Shifted Rosenbrock function after 250 generations of 50 samples each, optimized using the Differential Evolution algorithm.

Provided that the objective score is calculated as the sum of squared residuals, the rules for Metropolis acceptance are as follows:

- if the proposal is better than the parent, accept the proposal as the child;
- if the proposal is worse than the parent, calculate `alfa` as the proposal/parent ratio (`ratioPrPa`) raised to the power `-(1/2)*nObs`. `nObs` is the number of observations, or equivalently, the number of residuals. Next, draw a random number `Z` from the [0,1] interval using a uniform distribution.
  - if `alfa` is greater than `Z`, accept the proposal;
  - otherwise, re-accept the parent.

The nice property of Metropolis acceptance is that the frequency with which points get accepted is proportional to their probability (provided that you have sampled long enough for the parameter distribution to become stable). We will use this later to calculate the confidence interval associated with the prediction.

▷ 37. Adapt the acceptance scheme in 'runDiffEvoMetro.m' in accordance with the Metropolis rules and run the script.

▷ 38. Adapt your script in such a way that it stores the entire array `parents` for each generation in a new array `evalResults`. Note that `evalResults` has `nPop*nGenerations` rows and `nDims+1` columns and that it can be preallocated with `NaN` values using MATLAB's built-in `repmat` function.

▷ 39. Use the information from `evalResults` to re-run the last `nSets = 500` samples, but make sure to store the predictions associated with each parameter set as a row in a larger matrix `ySims` of size `nSets x nObs`.

▷ 40. Use the `prctile` function to calculate the 95% confidence interval and the median at each time step using `prctile(ySims,[2.5,50,97.5])`. Visualize the `prctile` results together with the observations `yObs`, and the `ySim` associated with the best-scoring parameter set.

▷ 41. You could also construct the 2.5% line by calculating the 2.5% quantile based on the last `nSets` parameter sets in `evalResults` (as opposed to the predictions associated with those sets) and run the model with that. Why is it generally incorrect to construct parameter intervals in this fashion?

`funcFlag`	**description**	**limits**
Unimodal Functions (5):		
1	Shifted Sphere Function	[-100,100]
2	Shifted Schwefel's Problem 1.2	[-100,100]
3	Shifted Rotated High Conditioned Elliptic Function	[-100,100]
4	Shifted Schwefel's Problem 1.2 with Noise in Fitness	[-100,100]
5	Schwefel's Problem 2.6 with Global Optimum on Bounds	[-100,100]
Multimodal Functions (20):		
Basic Functions (7):		
6	Shifted Rosenbrock's Function	[-100,100]
7	Shifted Rotated Griewank's Function without Bounds	[0,600]
8	Shifted Rotated Ackley's Function with Global Optimum on Bounds	[-32,32]
9	Shifted Rastrigin's Function	[-5,5]
10	Shifted Rotated Rastrigin's Function	[-5,5]
11	Shifted Rotated Weierstrass Function	[-0.5,0.5]
12	Schwefel's Problem 2.13	[-100,100]
Expanded Functions (2):		
13	Expanded Extended Griewank's plus Rosenbrock's Function (F8F2)	[-3,1]
14	Expanded Rotated Extended Scaffe's (F6)	[-100,100]
Hybrid Composition Functions (11):		
15	Hybrid Composition Function 1	[-5,5]
16	Rotated Hybrid Composition Function 1	[-5,5]
17	Rotated Hybrid Composition Function 1 with Noise in Fitness	[-5,5]
18	Rotated Hybrid Composition Function 2	[-5,5]
19	Rotated Hybrid Composition Function 2 with a Narrow Basin for the Global Optimum	[-5,5]
20	Rotated Hybrid Composition Function 2 with the Global Optimum on the Bounds	[-5,5]
21	Rotated Hybrid Composition Function 3	[-5,5]
22	Rotated Hybrid Composition Function 3 with High Condition Number Matrix	[-5,5]
23	Non-Continuous Rotated Hybrid Composition Function 3	[-5,5]
24	Rotated Hybrid Composition Function 4	[-5,5]
25	Rotated Hybrid Composition Function 4 without Bounds	[-2,5]

Table 2.1

Chapter 3

Working with DREAM

Aims:

1. to understand the functionality of DREAM;
2. to understand the basics of the implementation of DREAM;
3. to experience the process of parameter evolution;
4. to understand the implementation of a (linear) model in DREAM;
5. to experience the changes that need to be made when a new model is implemented in DREAM.

Before we can start optimizing the parameters for any model, we need to set up the DREAM algorithm (Vrugt et al., 2009). An efficient way of doing this is by creating a new m-file, that consists of 5 parts, which are used to:

1. clean up any old variables, figures etc. and add the DREAM folder to the MATLAB search path. This way, you can access the functions that make up the DREAM algorithm, even though they do not reside in the current working folder;
2. load or create any data that you will need during the optimization. This includes the observations but also the model constants, intial conditions, etc.;
3. specify the settings with which to run the DREAM algorithm. The most important variables that need to be initialized in this part are 4 structure arrays: `MCMCPar, ParRange, Measurement, Extra`;
4. call the DREAM algorithm;
5. visualize, analyze, and post-process the outcome of the optimization.

We have included an example of how DREAM can be used to calibrate the parameters of a linear model. The example is located at './exercises/dream-linear-model/'.

► 42. Use the MATLAB editor to open 'dreamWithLinearModel.m'.

In the DREAM settings part, the user typically adjusts the number of parameters (`MCMCPar.n`), the maximum number of model evaluations (`MCMCPar.ndraw`), the limits of the parameter space (`ParRange`), the measurements to be used in the objective function (`Measurement`), and of course the name of the function whose parameters are optimized (`ModelName`). Any additional arrays that the model needs to run—initial conditions, boundary conditions etc.–can be included as fields in the structure array `Extra`.

On the model side, DREAM has a few small requirements also; DREAM expects models to be formulated as a function with two input arguments, the first being a parameter vector of length `MCMCPar.n` and the second a structure array with additional data (initial conditions, boundary conditions, and anything else you need inside the function).

▶ 43. Open 'linearmodel.m' in the MATLAB editor to see how this works.

Note that DREAM compares the model output directly to the measurements stored in `Measurement.MeasData`, so they should have the same dimensions for this to work. Also, for the comparison to make sense at all, you need to make sure that the values in the model output and in the measurement correspond to the same point in time (or space). Models that use a variable time step can be especially tricky in this respect.

▶ 44. If you didn't already do this, run 'dreamWithLinearModel'.

During the optimization, you will see some figures being created automatically. Depending on which figures you want to see, you can set the `visualize*` variables in 'visDream.m' as you like. Furthermore, you can adjust the interval at which figures are plotted by setting `visInterval`, further down in 'visDream.m'.

▶ 45. Set your work directory to './exercises/dream-quadratic-model'. The model m-file in this directory is 'quadraticmodel.m'. Write a main script similar to 'dreamWithLinearModel.m', with which the parameters of the quadratic model are optimized. Copy and paste from the linear model example as appropriate.

▶ 46. After the optimization finishes, it's often desirable to save the variables and figures to harddisk, for example by:

```
% You can save the variables from the workspace with:
save('dream-results.mat','MCMCPar','Sequences','ParRange',...
                         'Extra','ModelName')

% and figures can be saved to a PNG file using:
figure(123)
print('model-prediction.png','-dpng','-r300')
```

This is especially useful for cases in which the model takes a bit of time to run.

Chapter 4

Objectives and information in data

Aims:

1. to be aware of the added value of carefully interpreting the results of the parameter evolution;
2. to understand the effect of noise in the measurements;
3. to understand the effect of the distribution of observations in relation to the model's local sensitivity (or system behavior);
4. to experience the added value of data transformations for parameter identification;
5. to understand the consequences of model structural errors for the parameter optimization process and result;
6. to experience the opportunities of multi-objective optimization.

Often people perform a model calibration, then they present the final model result, and then... they are done. But in fact the most interesting part of inverse modeling is the interpretation of both the results of the evolution of the parameter search and of the simulation results. In this chapter, we will investigate the relation between model complexity, measurement error, and the type of observations.

4.1 Case: cubic model

▶ 47. Use the MATLAB editor to open 'dreamWithCubicmodel.m' located in './exercise/dream-cubicmodel/'. After briefly studying the program, press F5 to run it as provided with 14 data points (variables `xObs` and `yObsNoisy`). Don't forget to save the results (check the code snippet on page 16 or copy and paste in a PowerPoint presentation).

▶ 48. To identify the parameters of a 4-parameter model, 14 data points aren't very many. How do you think the results will change if you increase the number of data points? Check your hypothesis by setting `xObsStep` to 0.2.

▶ 49. Not only the number of the data points is important, but also the distribution. Argue what would be the result if you wouldn't have had the data points below x=3 (keep `xObsStep` at 0.2). Explain which parameters you expect to be identified most accurately? Then check whether your hypothesis was correct.

▶ 50. If you could do one more measurement between x=-1 and x=7, where would you place it? Explain why.

▶ 51. What if you would have only the observations below x=3 (keep `xObsStep` at 0.2)? Again, explain which parameters you expect to be identified most precisely and then check whether your hypothesis was correct.

▷ 52. If you would increase the magnitude of the error term (`gaussTerm`), how will that affect the results? Check your hypothesis by making the appropriate changes to the script.

4.2 Case: rainfall interception

In this part, we will interpret the relation between data, model and selected parameters using a simplified version of an existing rainfall interception model (Bouten et al., 1996). This version simulates canopy water storage using one layer only. As with any model, it is important that you understand both the concept of the model—including equations, parameters and boundary conditions—as well as the measurements used in the inverse modeling; if you don't, it is impossible to interpret the results.

Conceptually, the interception model is a very simple water balance of the canopy. Water storage in the canopy (S) changes with time t, as a result of precipitation (P), interception (I), drainage (D) and evaporation (E); see Figure 4.1 and Equation 4.1.

$$\frac{\Delta S}{\Delta t} = I - D - E \tag{4.1}$$

with:

S	canopy water storage	mm
t	time	day
I	interception rate	mm·day^{-1}
D	canopy drainage rate	mm·day^{-1}
E	canopy evaporation rate	mm·day^{-1}

Precipitation and potential evaporation (E_0) are the measured boundary conditions. The other fluxes are calculated as:

$$I = a \cdot P \tag{4.2}$$

$$D = \begin{cases} b \cdot (S - c), & \text{if } S > c \\ 0, & \text{if } S \leq c \end{cases} \tag{4.3}$$

$$E = d \cdot E_0 \cdot \frac{S}{c} \tag{4.4}$$

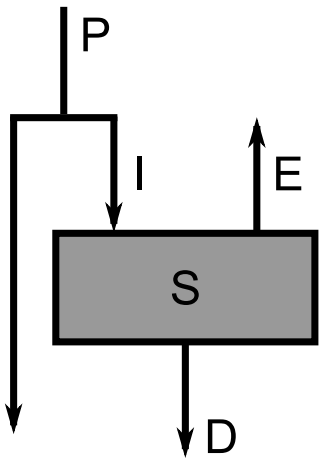


Figure 4.1: Definition of flows for the interception model.

parameter	**description**	**units**
a	interception efficiency (canopy cover fraction)	-
b	canopy drainage efficiency	day^{-1}
c	canopy water storage capacity	mm
d	evaporation efficiency	-

Before optimizing the model parameters a, b, c and d, you need to get a better understanding of the meaning of each parameter, and its effect on the model prediction. With this knowledge you will be able to choose a more meaningful minimum and maximum parameter space and your optimization will converge faster.

▶ 53. Use the MATLAB editor to open 'mancal_interceptionmodel' located in './exercises/manual-calibration-interception-model/'. After pressing F5 to run the script, a graphical user interface (GUI) will appear. If you run the model with the input values $a = 0.6$, $b = 200$, $c = 2.5$ and $d = 0.83$, you will get a plot similar to those in Figure 4.2. Explain the output of this simulation:

- why does the storage increase at $t = 0.75$?
- why does it reach a constant level?
- why is this level 2.6?
- why does this level change at $t = 1.0$?
- why does this level become 2.5?
- why does the storage decrease at $t = 1.25$?
- explain the shape of the storage curve after $t = 1.25$.
- draw the canopy evaporation rate in Figure 4.2.

▶ 54. Next we will alter the parameter values only one at a time (!) and predict how the change affects the model result. So, choose a parameter, then first draw your prediction in one of

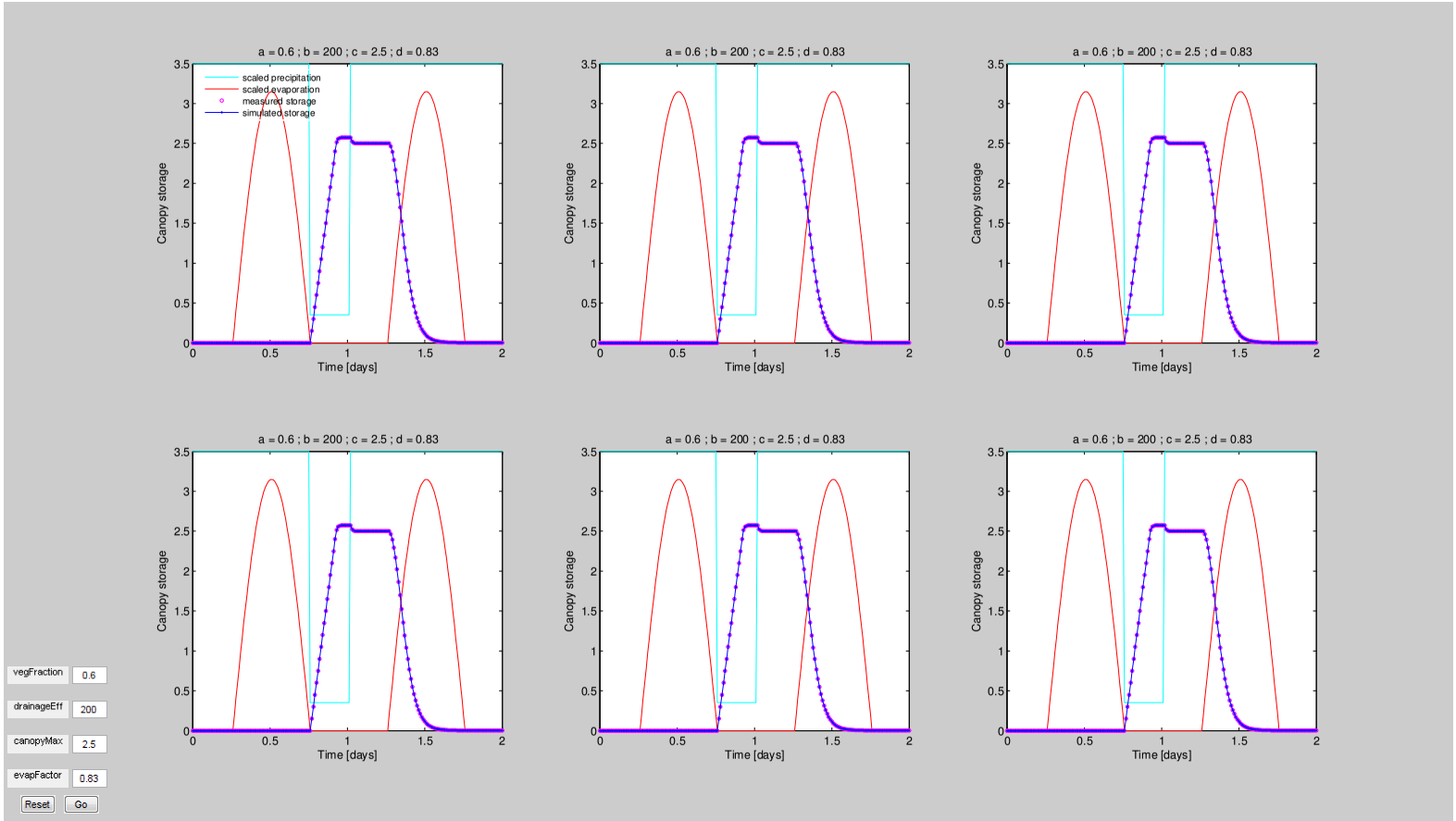


Figure 4.2: Graphical user interface showing the model prediction for $a = 0.6$, $b = 200$, $c = 2.5$ and $d = 0.83$, based on simplified boundary conditions.

the subplots of Figure 4.2. Only after you've drawn your prediction, change the parameter value in the GUI. Press the 'Go' button to run the interception model with the changed parameter. Try to explain why your prediction diverges from the model result. This is a useful procedure to better understand the behavior of a system.

► 55. Summarize the knowledge you gained about each parameter's effect on the model result.

4.2.1 Distribution of information within the data

Having done the manual calibration, you are ready to interpret the results of the interception model with boundary conditions that are more realistic. In the next few exercises, we will perform several DREAM calibrations of the interception model using different data sets.

As a first step, it is always wise to first check your inverse modeling design with so-called 'artificial measurements'. Artificial measurements are generated by running the model with a known parameter set. Subsequent calibration of the model to the artificial measurements should yield this same parameter set again.

► 56. Use the MATLAB editor to open 'dreamWithInterc.m' located in './exercises/dream-interception-model/'. By default this script uses the simplified boundary condition that we used in the manual calibration. Run the main script to check if the true parameters are found.

► 57. Now adapt your script to let it use the more realistic boundary conditions, by uncommenting the appropriate parts in the script. Experiment with different boundary conditions. Can you explain the differences between the various data sets? And what is the difference between the realistic boundary conditions on the one hand and the ideal case on the other?

It is often interesting to calibrate your model parameters based on a subset of the data. This way, you can get some idea of the type of information that is needed to identify each model parameter with precision and accuracy. The generic way to have DREAM use subsets, is by adding a field to `Extra` that contains the observation times that you want to use as part of your objective function:

```
Ix = find(stor(:,1)<1.0);           % find the rows of the array 'stor' where
                                    % time is less than 1.0

Measurement.MeasData = stor(Ix,2); % Define the measured data

Extra.bound = bound;                % these are the boundary conditions (t,P,PE)
Extra.stor = stor;                  % these are the measurements of storage (t,S)
Extra.tObj = stor(Ix,1);            % the times at which measurements are available
                                    % that are used in the objective function
```

Inside `interceptionmodel`, you can use the time information to only export simulated storage values at the times specified in `tObj`:

```
output = interp1(TimeTab,Storage',Extra.tObj);
```

`output` can subsequently be returned as the function's output argument, after which DREAM will compare it to the contents of `Measurement.MeasData`.

After the optimization finishes, the script probably throws an error in the visualization part. This is because the simulated time series of storage now has less elements than `bounds(:,1)`. You can avoid this by re-defining `Extra.tObj` just before re-running the last 1000 parameter sets:

```
parCols=1:MCMCPar.n;
lastRows = [-99:0]+size(Sequences,1);
parSets = [];
for iSeq=1:MCMCPar.seq
    parSets = cat(1,parSets,Sequences(lastRows,parCols,iSeq));
end

Extra.tObj = bound(:,1);
for k=1:size(parSets,1)
    parVec = parSets(k,:);
    ySim(k,:) = interceptionmodel(parVec,Extra);
end
```

► 58. Adapt your script to let it use only the observations for which $t < 1$ (simplified boundary conditions). Make sure that the storage values returned by `interceptionmodel` refer to the times at which storage was observed. What do you expect with regard to the identifiability of the parameters? Store the results to be able to refer back to them later, for instance by copying and pasting into a PowerPoint presentation.

► 59. In the previous exercises, we have calibrated to a time-based subset of the observations. However, you can also make a subset based on certain characteristics of the data. For instance, make a subset based on $S > 2.2$ mm (use the '211' data), calibrate the interception model and explain the results. Remember, you can add fields to `Extra` with information you need inside your model's workspace. For instance, you could do:

```
Ix = find(stor(:,2)>2.2);           % find the rows of the array 'stor' where
                                    % storage is more than 2.2

Measurement.MeasData = stor(Ix,2); % Define the measured data

Extra.bound = bound;                % these are the boundary conditions (t,P,PE)
Extra.stor = stor;                  % these are the measurements of storage (t,S)
Extra.tObj = stor(Ix,1);            % the times at which measurements are available
                                    % that are used in the objective function
```

► 60. If you didn't already do this, adapt the script such that the simulated canopy storage dynamics are visualized starting from t=211.41 through t=212.65. Let your visualization differentiate between observations that are part of the objective set and those that aren't. You should get a figure more or less like Figure 4.3.

► 61. Describe the characteristics of a minimal data set with which the parameters of the interception model can still be identified. Does the measurement accuracy affect the minimal set?

4.2.2 Data transformations

In this part, we will experiment with using different kinds of transformations. For instance, squaring the observed data (Y^2) emphasizes the high values, whereas $\sqrt{Y}$ and Box-Cox transforms ($\frac{Y^\lambda - 1}{\lambda}, \lambda = 0.3$) give more weight to lower (non-negative) values. Also, Box-Cox transformations are often used when the error is known to be heteroscedastic (i.e. when the magnitude of the error is dependent on measurement value). $log(Y)$ and $log_{10}(Y)$ should be

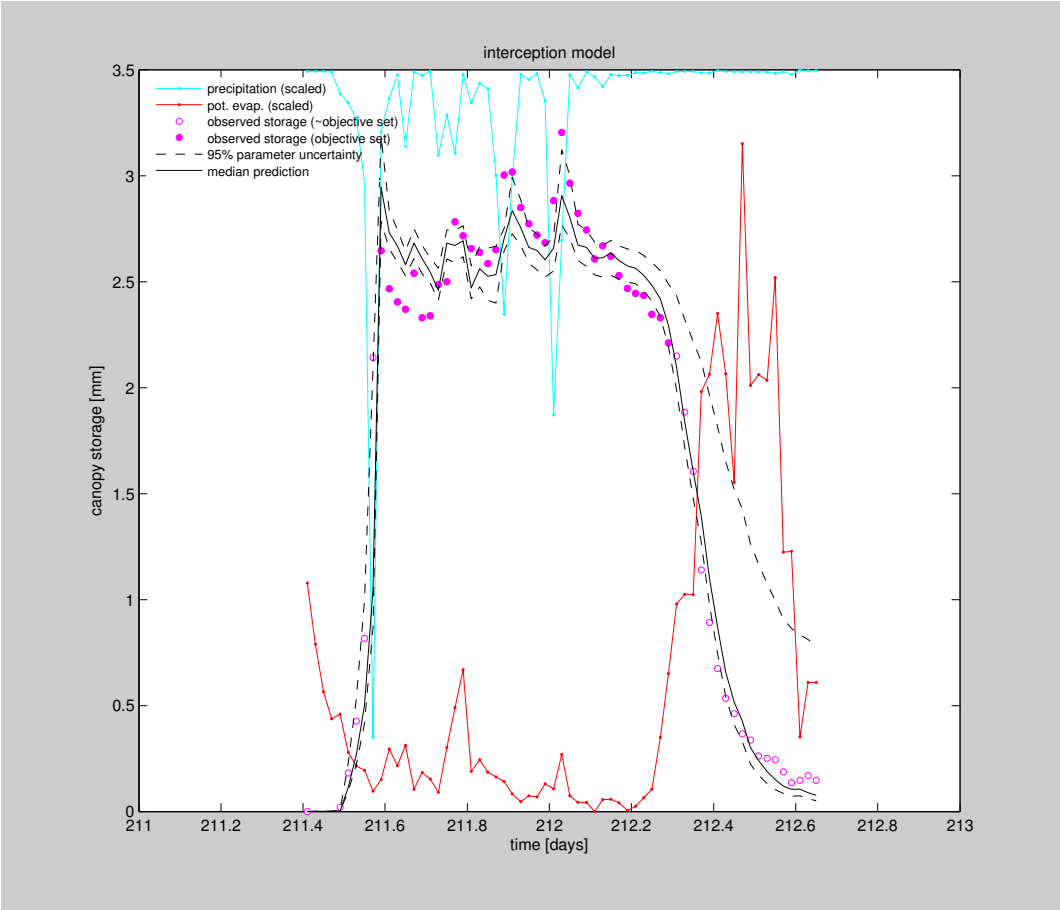


Figure 4.3: Example of the interception model with objective based on high-storage values only.

avoided if Y-values of 0 can occur.

▶ 62. Use the MATLAB editor to open 'dreamWithHymodBoxCox.m' located in './exercises/dream-hymod-box-cox'. Despite its name, the script does not perform a Box-Cox transform yet, but we'll get to that in a minute. First run the script as provided and interpret the results.

▶ 63. What change in model performance or identifiability of parameters would you expect if you apply the Box-Cox transform? Before running the script, we need to make some small changes first. Of course, you need to transform the observations as well as the model output. In order to visualize the optimization results, we need to adapt the model such that it can output either the untransformed simulated storage or the transformed simulated storage, depending on a flag `Extra.argOutIsTransformed` which can be `true` or `false`. For instance:

```
if Extra.argOutIsTransformed
    output = transformedOutput;
else
    output = normalOutput;
end
```

This is an easy way to use transformed data for the optimization, and untransformed data for the subsequent visualization. Make the appropriate changes in the program and check the answer you gave earlier.

Another interesting transform is to use the derivative of the observations. This can be particularly helpful with determining rates with respect to time.

▷ 64. What result do you expect if you apply the derivative transformation in case of the cubic model?

▶ 65. Use the MATLAB editor to open 'dreamWithIntercDerivative.m' located at './exercises/dream-interceptionmodel-derivative/'. As you can see, the objective function operates on the derivative of storage. How do you think the prediction will be different from the earlier results? Check your answer by running the script.

4.3 Case: Easter Island population dynamics

In the past few exercises we have seen that successful parameter identification depends on what we call "the information content of data". It depends on:

1. the signal/noise ratio (the higher the ratio, the faster the convergence of the search procedure);
2. the sensitivity of the model for the parameters;
3. the distribution of the measurements in relation to this sensitivity.

Keep in mind that a model is never perfect and that deviations between model output and observations is not by definition a Gaussian distribution. If the model structure causes systematic errors, the inverse modeling procedure as used until now will lead to a best fit

while tuning the parameters. In this case the optimal parameter values will compensate for the model structure error. In such a case parameter identification produces the "wrong" (but optimal) parameter values and convergence will also become slower.

To explore this effect in more detail, we will test a model of human population and resources (trees/biomass) of Easter Island in the period 0-2000 AD. The model is based on Brander and Taylor (1998). The governing equations and probable parameter values are as follows.

Rate of change of population:

$$\frac{dP}{dt} = (b + cr \cdot \frac{C}{P}) \cdot P - (d - cr \cdot \frac{C}{P}) \cdot P \tag{4.5}$$

Rate of change of resources:

$$\frac{dR}{dt} = g \cdot (1 - \frac{R}{Cc}) \cdot R - C \tag{4.6}$$

Consumption rate:

$$C = p \cdot P \cdot R \tag{4.7}$$

with:

	variable name as used in the model	**description**	**probable value**
P	`Pop`	human population	
R	`Res`	resources (biomass)	
b	`parPopGrowth`	relative birth rate	0.02
d	`parDeathRate`	relative death rate	0.03
Cc	`parCarCap`	carrying capacity	160
cr	`parConsResponse`	consumption response	150
g	`parRelResGrowth`	relative growth rate of resources	0.004
$cr \cdot \frac{C}{P}$	`consProfit`	consumption profit	
p	`parFit`	labor efficiency constant	0–1e-6

▶ 66. Use the MATLAB editor to open 'dreamWithEasterModel.m' from './exercises/dream-easter-island/'. Figure 4.4 shows 5 data sets that we have created using the model described above. Experiment with calibrating the `eastermodel` based on either population or resources data from any of the files listed below. Assume an initial Population of 20, and an initial Resources value of 160. To help the interpretation of the results, here are a few pointers on how the data were created:

1. 'pop-res-data-ref.mat' is the reference file that contains the artificial observations generated with the parameters mentioned above, with only very little noise superimposed;
2. 'pop-res-data-ref-noise.mat' is the same as (1), but with more noise;
3. 'pop-res-data-variable-deathrate.mat' is a file that contains the artificial observations generated with the parameters mentioned above, except that the death rate parameter was changed to 0.035 for $t > 700$. Because the `eastermodel` does not incorporate this change, a structural error is present in the model. Also, there's only very little noise superimposed;
4. 'pop-res-data-variable-deathrate-noise.mat' is the same as (3), but with more noise.
5. 'pop-res-data-high-deathrate.mat' is the same as (1), but with `parDeathRate=0.035`.

▶ 67. Explain the high correlation between `parDeathRate` and `parPopGrowth` in the results of the calibration.

▶ 68. What is the effect of the higher noise in the observations?

▶ 69. Set the value of `parPopGrowth` to 0.02 (i.e. exclude it from the optimization) to avoid the high correlation between `parDeathRate` and `parPopGrowth`. How do you think this will affect the optimization process and the accuracy of the optimized parameters (in comparison to the former case)? Evaluate the result. Is this what you expected? If not, explain the difference.

▷ 70. If you want to play with this some more, you can make your own artificial data using 'eastermodelArtifMeas.m'.

So far, we have calibrated the `eastermodel` based on either Population or Resources, but you could also use both variables simultaneously. To do this, we need to make some changes to the model:

```
% Interpolate the simulated Population and Resources to the times
% at which measurements are available against which to compare the
% simulations:
PopSimI = interp1(tSim,PopSim,tMeas);
ResSimI = interp1(tSim,ResSim,tMeas);

% For each data series, calculate the absolute error:
ErrPop = abs(PopSimI-PopMeas);
ErrRes = abs(ResSimI-ResMeas);

if Extra.argOutIsErr
    % For the optimization, use the weighting factor 'Extra.popWeight'
    % to calculate a weighted error:
    OUT = Extra.popWeight*ErrPop+(1-Extra.popWeight)*ErrRes;
else
    % Or, for the post-optimization part, output the time series of
    % Population and Resources:
    OUT = [PopSim,ResSim];
end
```

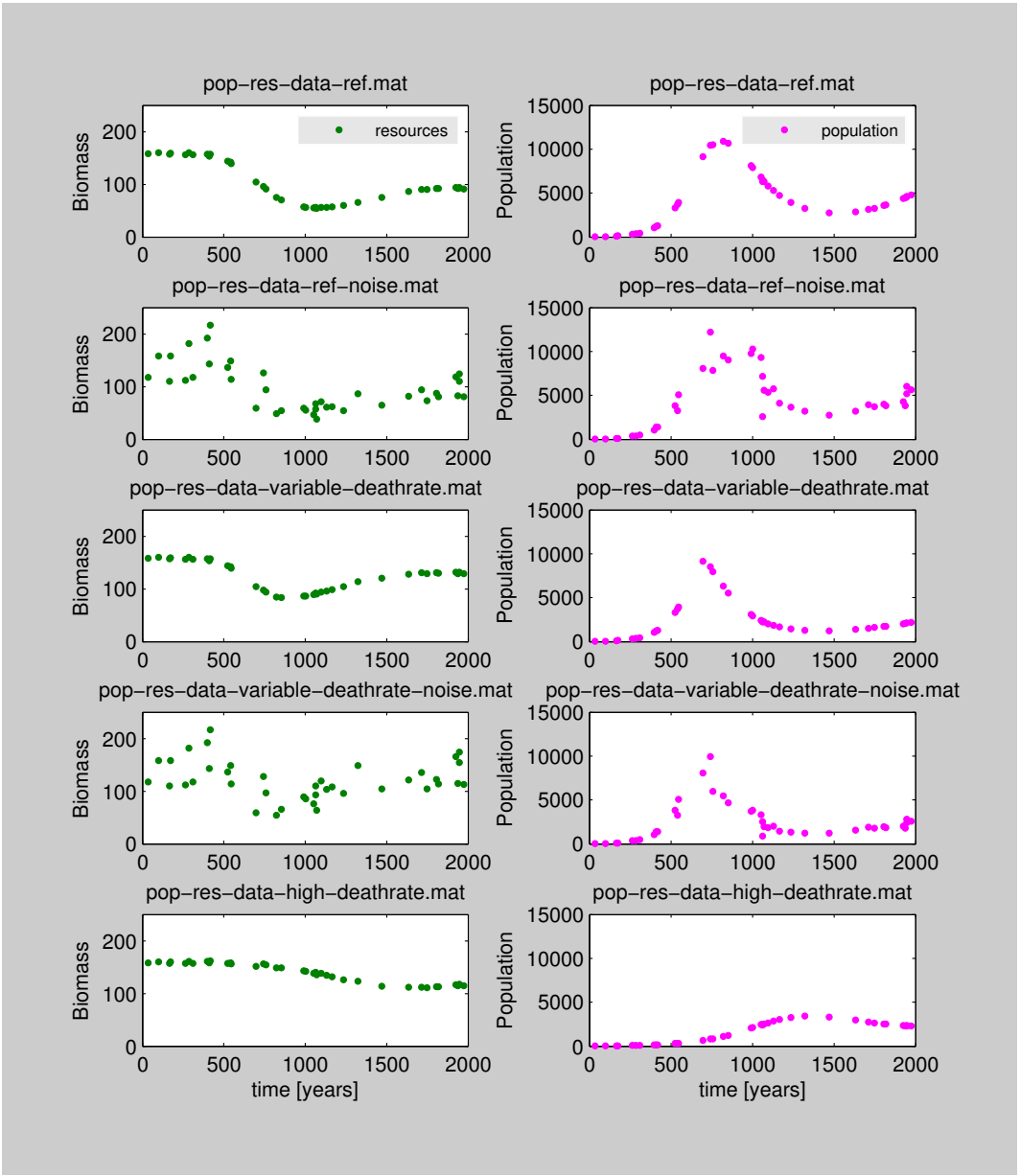


Figure 4.4: Resources-Population plots for the various files.

With this code, the model returns an error series, rather than the states themselves, so the objective function should no longer compare model output with observations; instead, we now want to minimize the model output. A simple way to accomplish this is by setting `Measurement.MeasData` to `zeros(size(PopMeas))`.

▶ 71. Use the MATLAB editor to open 'dreamWithEasterModelMO.m' from './exercises/dream-easter-island-mo/'. Make the appropriate changes to the program in order to use both Population and Resources information. Set `Extra.popWeight` to 1.0 for now (i.e. ignoring the misfit on Resources) and don't forget to initialize `Extra.argOutIsErr` as well. With this weighting factor, your result should be similar to that of a single-objective optimization (when the objective function operates on Population). Run the script. Store your results.

▶ 72. Now do the same thing, but with all weight on the Resources. Again, store the result.

▶ 73. Repeat one more time, but with half the weight on the Population and half the weight on Resources. Store the result. Explain why the last result looks much more like the result of `Extra.popWeight=1.0` than that of `Extra.popWeight=0.0`.

▶ 74. How would you make sure that Population and Resources are treated as equally important variables when `Extra.popWeight=0.5`? Implement your solution and check if there is any difference with your earlier results.

▶ 75. Adapt the script to let it use the Resources data from 'pop-res-data-ref.mat' and the Population data from 'pop-res-data-high-deathrate.mat' with:

```
dataFileRes = 'pop-res-data-ref.mat';
dataFilePop = 'pop-res-data-high-deathrate.mat';

% load only 'PopMeas' from 'dataFilePop':
load(dataFilePop,'PopMeas')

% load 'ResourceMeas' and 'tMeas' from 'dataFileRes':
load(dataFileRes,'ResourceMeas','tMeas')
```

Run the script for `Extra.popWeight` equal to 1.0, 0.0, and 0.5. Explain the results.

▷ 76. Looking back at the single-objective exercises that we have done earlier this week, which one do you think would benefit from a multi-objective approach? Explain what variables you would use in the calibration and which objectives you would use (and why) before implementing it.

4.4 Multi-objective optimization of HYMOD

▷ 77. In the directory './exercises/hymod-so/' you will find the files to apply the SCE-UA algorithm (Duan et al., 1992) to the HYMOD model. We use the same data as in previous exercises (Leaf River, Mississippi), and we analyze the data from a single year (1953). Open 'CompOF.m' and verify what objective function is being used in the current optimization. In a first step, compare the root mean square error (RMSE) and the mean absolute error (MAE). Run SCE-UA and save the best parameters and the associated model runs (BestSim and bestSets) for both objective functions. Note that the objection function values for RMSE and MAE are given at the end of the SCE-UA run. Are the optimized parameters different for the two objective functions? Are the model predictions different? Can you understand why?

▷ 78. If you like, you can implement alternative objective functions. If you need inspiration, you might want to take a look at Table 1 in Gupta et al. (1998); see the './papers/' folder. Alternatively, you can apply a Box-Cox transformation to the data. Again, can you understand why the model parameters and model predictions are different? Is one of the optimized parameter sets superior to another one?

▷ 79. The Pareto front can be approximated by varying the weights a and b in the following objective function: $a*RMSE+b*MAE$. If $a = 1$ and $b = 0$, you should get the same result as for the single-objective optimization with RMSE as the objective function. Implement this weighted objective function in `CompOF` and run the SCE-UA algorithm with different selections for the parameters a and b. Were you able to systematically explore the Pareto front? Is this an efficient approach?

▷ 80. In order to explore the Pareto front between the RMSE and the MAE more accurately, we will apply the MOSCEM-UA algorithm to HYMOD. You can find the code in the directory './exercises/hymod-mo/'. The two objective functions are defined at the end of `hymod`.

▷ 81. You can start the algorithm by typing `runMOSCEM` at the command prompt. Compare the results of MOSCEM-UA with those of the previous exercise. Does MOSCEM-UA provide a good estimate of the Pareto front? Do the optimized parameters vary systematically along the Pareto front? Can you draw any conclusions about the model from this? Explore different objective functions.

▶ 82. Formulate 2 conclusions on the basis of today's activities:

1. your most important conclusion;
2. a conclusion based on your findings, that you don't expect anyone else has found.

Bibliography

Bouten, W., M. G. Schaap, J. Aerts, and A. W. M. Vermetten (1996). Monitoring and modelling canopy water storage amounts in support of atmospheric deposition studies. *Journal of Hydrology 181*, 305–321.

Boyle, D. P., H. V. Gupta, and S. Sorooshian (2000). Toward improved calibration of hydrologic models: Combining the strengths of manual and automatic methods. *Water Resources Research 36*(12), 3663–3674.

Brander, J. A. and M. S. Taylor (1998). The Simple Economics of Easter Island: A Ricardo-Malthus Model of Renewable Resource Use. *The American Economic Review 88*(1), 119–138.

Duan, Q., V. K. Gupta, and S. Sorooshian (1992). Effective and efficient global optimization for conceptual rainfall-runoff models. *Water Resources Research 28*(4), 1015–1031.

Gupta, H. V., S. Sorooshian, and P. O. Yapo (1998). Toward improved calibration of hydrologic models: Multiple and noncommensurable measures of information. *Water Resources Research 34*(4), 751–763.

Nelder, J. A. and R. Mead (1965). A simplex method for function minimization. *The Computer Journal 7*(4), 308–313.

Storn, R. and K. Price (1997). Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization 11*, 341–359.

Vrugt, J. A., C. ter Braak, C. Diks, B. A. Robinson, J. M. Hyman, and D. Higdon (2009). Accelerating Markov Chain Monte Carlo Simulation by Differential Evolution with Self-Adaptive Randomized Subspace Sampling. *International Journal of Nonlinear Sciences & Numerical Simulation 10*(3), 271–288.