Research internship

Developing a prototype of Bayesian Inference framework to recalibrate the complex hydrological model LARSIM

Department of Informatics

Supervisor: Univ.-Prof. Dr. Hans-Joachim Bungartz

Co-Supervisor: Ivana Jovanovic Buha, M.Sc. (hons)

Document handed in to supervisor on the 28th of September 2020.

Supervisor's signature: Stamp of the department:

Mathieu Putz

Matrikelnummer: 03716001

1. Abstract

Bavaria uses the LARSIM hydrological model to simulate and predict water levels. This model is currently being recalibrated based on over fifteen years of historical data. In this internship, a Metropolis-Hastings Markov Chain Monte Carlo (MCMC) simulation was developed, with the goal to ultimately help find parametrizations that perform better on past data. With this code having been written, the next step will be to parallelize the program, such that it can run a greater number of simulations and exploit more historical data.

2. Introduction

LARSIM or the Large Area Runoff Simulation Model is a highly complex hydrological model, that is being used by the state of Bavaria to predict water levels. This helps locals to prepare for droughts and floods. For the purposes of this internship, we focused on the water level at a single station. Not only does that make the problem more tractable, it also makes sense, since the chosen station is downstream of all the others, after many streams have flown together. Hence it captures a nice synopsis of regional water levels. The model relies on dozens of parameters. Some of them have direct physical interpretations, others are merely conceptual. With this model having been used for over fifteen years, there is now ample data available on its historical performance. Ivana Jovanovic Buha is currently working on recalibrating the parameters based on this data. It was my task to develop a particular kind of Bayesian inference model for this problem, a Metropolis-Hastings Markov Chain Monte Carlo (MCMC) simulation. Because of computational constraints, the dimensionality of the problem must be reduced. Instead of optimizing the dozens of parameters, the simulation is intended to find values for about four critical parameters that are near-optimal when other parameters are kept constant. If computational constraints were eased, the number of optimized parameters could be increased with minimal changes to the program.

However, the MCMC algorithm is inherently sequential. Even with only four parameters, the LARSIM model is too expensive to run it many hundreds of times without applying parallelization techniques. That task was beyond the limited scope of this internship. Hence an opportunity for future research: applying parallelization techniques to the developed prototype.

3. Theoretical background

Bayes' theorem describes how a perfectly rational agent updates their beliefs in light of new evidence, while accounting for prior knowledge. In its simplest form, it reads:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Where P(A) and P(B) describe the probability of event A and B respectively and P(A|B) and P(B|A) describe the probability of A given B and vice-versa. P(A) is commonly called the *prior*, i.e. the probability that we assigned event A prior to seeing evidence B.

Models that are based on Bayesian inference use this formula to evaluate the likelihood of different hypotheses being true. In our case, a hypothesis states that a certain parametrization of LARSIM is the "true" one.

The prototype developed in this internship is a Metropolis-Hastings Markov Chain Monte Carlo (MCMC) simulation, which is a particular kind of Bayesian inference model. Generally, a Monte Carlo simulation is a numerical method using repeated random sampling to find an approximation to a particular quantity in settings too complex to allow for an analytical solution. Here, parametrizations to be tested are drawn stochastically from certain distributions. Markov Chains have the characteristic that the probability of an event depends only on the event just preceding it. For the purposes of this project, that means which parametrization will be tested next depends only on the previously accepted parametrization. With this understanding, we can now proceed to explain the Metropolis-Hastings MCMC algorithm and what it means in the context of this project.

As stated above, the fundamental goal is to find values for a few parameters that best fit the historical data, when other parameters are kept constant.

At the beginning we might have some idea what values the parameters should approximately take. This is encoded in our *prior (notation: P(parameters))*, a probability density function that describes how likely different values of the parameters would be based on our knowledge before running the algorithm. The prior will be taken into account when considering parameter choices. A combination of parameters that performs very well on historical data, will be disadvantaged if it conflicts with our prior beliefs about what values the parameters are likely to take, for instance for reasons related to their physical interpretation. In our case, we chose uniform priors within a given range and a prior of zero outside of that range.

How well the parameters perform on historical data is measured by the *likelihood function (notation: P(parameters|data))*. It expresses how well the results obtained by running LARSIM with a given set of parameters fit the historical data, without considering our prior. We chose to represent the likelihood of different parameter distributions with a Gaussian, whose standard deviation was chosen to be equal to that of the historical track record. Here each data point is treated individually and then the probabilities are all multiplied together, to obtain the likelihood. This assumes that errors are independent of each other, which is of course untrue. If the model is wrong about today, it will likely also be wrong about tomorrow. However, any alternatives would add a lot of complexity, hence this assumptions will tentatively be used as the algorithm should still produce useful results.

The algorithm also involves randomly generating a new set of parameters in each iteration, based on a certain probability density function, called the *proposal distribution*. Importantly, this distribution is centered around the last accepted set of parameters, which makes it a Markov Chain. Again, we chose a Gaussian for each parameter; the standard deviation was provisionally set to one twentieth of the range. While the choice of the standard deviation does not impact how the algorithm fundamentally works, it does impact how fast it converges. In later applications, experiments should be used to determine which standard deviation makes the algorithm converge the fastest. It will probably be easiest to test this with low dimensionality, also analyzing whether dimensionality has an impact and adjusting the values accordingly for high dimensionality simulations.

We may now look at the first iteration of the actual algorithm. It starts off with an initial guess of what the optimal values for the parameters might be. The choice of these values does not matter much; the algorithm should converge to the same optimum irrespectively. Then, the algorithm randomly generates new values for the parameters based on the proposal distribution, which favors values in the vicinity of the current value. Next, it runs LARSIM with both sets of parameters, to obtain the likelihood functions for each. The two sets are compared to each other based on both the likelihood function and the prior. More on that below. If the new set of values is better than the old one, it is accepted. Otherwise it is randomly accepted some fraction of the time and rejected the rest of the time. If it is much worse, it will very likely be rejected, if it is close, it may well be accepted. This helps avoid local optima. Finally, the process repeats itself and the next randomly generated set of parameters will be compared to this new set if it was accepted or the old one otherwise. We will store all the accepted and rejected values in separate arrays. After running the desired number of iterations, the array of accepted sets of parameters will contain the information we sought: we can take an average of the values in that set. To eliminate the influence of our initial choice of parameters, we will discard the first few iterations. These are commonly referred to as burn-in. After this initial phase, the algorithm should start converging, with all the accepted sets of parameters scattered around an optimum.

The function evaluating whether a given set of parameters is better or worse than another one is the following:

$$\alpha = \frac{P(data \mid new\ Parameters) \cdot P(new\ Parameters)}{P(data \mid old\ Parameters) \cdot P(old\ Parameters)}$$

If α is bigger than one, the new set is always accepted, if it is between zero and one, the new set is accepted with probability α.

The equations in the code below result after taking the logarithms of both sides of this equation. This has the advantage, that products of small probabilities are converted to sums of large negative numbers. This is helps avoid computational errors, while making no difference mathematically.

Note that the described algorithm is inherently sequential. Each new set of parameters is randomly generated based on the one previously accepted and then compared to that set.

To allow for parallelization, which would bring computational advantages, the algorithm will need to be tweaked.

4. Code explanation

This section shall discuss the code itself.

The beginning of the program mostly sets up LARSIM. In addition to that, values are chosen for certain parameters, like the standard deviations mentioned above. Skipping these sections, we will focus on the core of the program here.

The following lambda function defines the proposal distribution (a.k.a. transition model). It uses the numpy library to draw random values from a gaussian distribution, centered around the previous value of that parameter and with the standard deviation a constant defined above in the program (as stated above, we used one twentieth of the range for this prototype).

```
transition_model = lambda x:      [np.random.normal(x[0],stddevParameters[0]),
                                   np.random.normal(x[1],stddevParameters[1]),
                                   np.random.normal(x[2],stddevParameters[2]),
                                   np.random.normal(x[3],stddevParameters[3]) ]
```

Next, a function computing the prior is defined. The array x contains the set of parameters to be evaluated. As stated above, we chose a uniform prior if all the parameters are within certain ranges and a prior of zero if one is outside that range. Given that we use logarithms, a prior of zero will later be converted to negative infinity since $\log(0^+) = -\infty$. This makes it impossible for that set of parameters to get accepted. If all the parameters are inside their respective range, we can use any constant, since it cancels out of the calculation due to the uniform prior; we arbitrarily chose to return one, which makes the calculations somewhat easier since log(1)=0.

```
def prior(x):
    acceptable = True
    for idx, p in enumerate(x):
        if (p<limits[idx][0] or p>limits[idx][1]):
            acceptable = False
            break
    if(acceptable):
        return 1
    return 0
```

To compute the logarithm of the likelihood function, the following function was defined. It receives two arguments; predictions contains all the predictions made by LARSIM for the

water levels in a given time period and depends on which parameters were used to run LARSIM; data contains the ground truth, the measured water levels. After verifying that the two arrays have the same lengths, the function returns log(P (data | parameters)).

P(data | parameters) is the product of the likelihoods of each datapoint, which in turn is given by a Gaussian distribution as stated above. Trivial algebraic transformations lead to the following code.

```
def manual_log_like_normal(predictions,data):
    n = len(predictions)
    if len(data)!=n:
        raise Exception("data and predictions have different lengths")
    return n*( -0.5*np.log(2*np.pi) - np.log(stddev) ) + np.sum( - ((predictions-
data)**2) / (2 * stddev**2) )
```

Finally, we need to define one more function before getting to the heart of the program, i.e. the acceptance rule, deciding whether to accept or reject a set of parameters after having computed the likelihood function and the prior. Two arguments are handed to the function; the first is the sum of the logarithm of the likelihood function and of the logarithm of the prior (either zero or negative infinity) of the previous set of parameters, the second is the same quantity for the new set of parameters. As explained above, the parametrization is always accepted if $\alpha \geq 1$ (if clause) and accepted with probability α otherwise (else clause). To compare the randomly generated number between 0 and 1 to the difference of the logarithms, the latter has to be exponentiated.

```
def acceptance(xLogLikelihood,newxLogLikelihood):
    if newxLogLikelihood>xLogLikelihood:
        return True
    else:
        accept=np.random.uniform(0,1)
        return (accept < (np.exp(newxLogLikelihood-xLogLikelihood)))
```

Finally, we are ready to discuss the implementation of the Metropolis-Hastings MCMC algorithm. It receives six arguments.

- likelihood_computer will be the manual_log_like_normal function above.
- prior will be the prior function above.
- transition_model will be the lambda function transition_model above.
- param_init is an array containing the initial guesses for each parameter. As explained above, this choice does not significantly affect the outcome if we run enough iterations. We chose the geometric mean of the lower and upper range bounds for each parameter.
- iterations is an integer containing the number of iterations we wish to run.
- acceptance_rule contains the acceptance function above.

The algorithm returns two arrays: accepted and rejected. They respectively contain the accepted and the rejected sets of parameters in order.

I left out two complicated but uninteresting sections of code and described what they do within the curly braces. x is the array of current parameters and x_new is the array of the new suggested parameters, which can either be accepted or rejected. dataCurr and dataNew contain the ground truth measurements for the period in question. They should be equal at all points in time; however, they were kept separate to make sure they each contain entries for the exact same dates as the respective predictions. These in turn are stored in the arrays predictionArrayCurr and predictionArrayNew. x_lik and x_new_lik are the likelihoods (i.e. without prior) of each set of parameters as estimated by our likelihood function. With these clarifications and with the above explanation of the algorithm the following code should now be understandable.

```
def metropolis_hastings(likelihood_computer, prior, transition_model, param_init,
iterations, acceptance_rule):

    x = param_init

   { section of code that

      1) runs LARSIM, extracts the relevant predictions and stores them in the
         array predictionArrayCurr
      2) stores the ground truth data for the same period in dataCurr

      }

    accepted = []

    rejected = []


    for I in range(iterations):

        print("#####New Iteration. Number ", I, " ########")

        x_new = transition_model(x)

      { section of code that

         1) runs LARSIM, extracts the relevant predictions and stores them in the
            array predictionArrayNew
         2) stores the ground truth data for the same period in dataNew
         }


        x_lik = likelihood_computer(predictionArrayCurr, dataCurr)

        x_new_lik = likelihood_computer(predictionArrayNew, dataNew)


      if (acceptance_rule(x_lik + np.log(prior(x)), x_new_lik +

          np.log(prior(x_new)))):

            x = x_new

            accepted.append(x_new)

            predictionArrayCurr = predictionArrayNew
```

```
            dataCurr = dataNew
            print("New value was accepted")
        else:
            rejected.append(x_new)
            print("New value was rejected")
    return np.array(accepted), np.array(rejected)
```

5. Next steps

The problem at hand is high-dimensional, complex and involves significant amounts of data. Hence, parallelization is essential to obtain reliable results. Devising a parallelized version of MCMC is the next step in the pursuit of recalibration; in that process the developed code will serve as a useful basis. Furthermore, it will be helpful to rely on existing methods to parallelize MCMC (e.g. Calderhead (2014)).

6. Sources

While Metropolis-Hastings MCMC is a widely used standard method, this implementation was inspired by the following source:

https://github.com/Joseph94m/MCMC/blob/master/MCMC.ipynb

A method to parallelize MCMC can be found in:

Calderhead, B.: A general construction for parallelizing Metropolis–Hastings algorithms. Proc. Natl. Acad. Sci. USA 111, 17408–17413 (2014)