I thank the authors for their detailed revision of the manuscript. Overall, their revision helped clarifying a lot of uncertain aspects of the algorithm, and their responses to my concerns were mostly satisfactory, but work still remains to be done. Unfortunately, I fear that my main concerns about the theoretical foundations and consequences of the error re-ordering remain inadequately addressed. The derivation provided in Appendix B does not sufficiently clarify these points eithers: the core question (*how exactly the re-ordering affects the base error distribution's statistical moments or hyperparameters, and what the consequences are in Bayesian terms*) remains unaddressed.

Since I take it that the authors would like to stick with the error re-ordering approach, I would argue that this leaves you with two possible pathways:

1) Provide a thorough theoretical derivation and in-depth investigation of what effect the error re-ordering really has, and what this means in Bayesian terms. After experimenting a bit with error re-ordering myself (see the Python code snippet below), I believe that a good start point might be couplings or measure transport (Pierre E. Jacob has a nice online lecture series on that called *Couplings and Monte Carlo*), as you seem to convert one distribution (the raw error distribution) into one with different statistical moments, one somehow moulded to the ideal error realizations.

2) Alternatively, you could simply drop the "Bayesian" attribute from your study or replace it with "Pseudo-Bayesian". This might require that you adjust the acronym. Even without full theoretical justification, your algorithm can still provide a useful heuristic, and maybe that is enough. Even in this case, however, I believe the manuscript would benefit from an isolated analysis of what mechanically happens to the input error distribution when subject to reordering. I have provided a few thoughts on this below.

As a consequence, I would recommend another round of major revisions. If you follow option (2), I recommend specifically to add a new section into the theory/methodology chapter which explores and illustrates the effects of re-ordering errors on the raw error distributions in detail (this is important for the reader's understanding of the approach– it should be part of the main manuscript, not the Appendix). To make space for this section, you could absorb some of the practical comments in the discussion (specifically, sections 4.1 and 4.3). There are also a lot of tangential comments addressing reviewer concerns throughout the manuscript which could be removed if their key points are addressed in this new section. This might also support the narrative thread of the manuscript by helping you to avoid the need to go on explanatory tangents.

As I don't want to leave you hang out to dry on such a large and amorphous task, I would specifically suggest exploring a simple example case in this proposed section. Specifically:

- Ignore the model (for the purpose of error reordering this is unnecessary); instead, skip straight to positing some hidden "ideal" sequence of error realizations which perfectly compensate the true residual error (similar to your figure A1); derive the corresponding ranks;
- Use a simple, structured residual error sequence to make it easier to read the induced effect. The sequence doesn't have to be realistic, merely insightful;
- Use a residual error distribution perfectly adjusted to the structured error you defined. In a synthetic test case, it's easy to derive an empirical cdf.
- Then explore the consequences of re-ordering for time series of different length or input error distributions of different quality. Discuss the statistical moments after reordering.

I have provided an example Python script for this below and appended some of its result figures for different time series lengths of 10, 100, and 1000 in Figures 1, 2, and 3. In this example, I arbitrarily assumed the true/ideal error realizations to follow a sine curve. Note that something more realistic (like random samples from a Gaussian distribution) would have also worked, but the simplicity of a sine curve makes it significantly easier to read the effect of the re-ordering.

Some thoughts on the results:

As I suspected in major comment #7 for the first round of revisions, the longer the time series, the more likely the method is to achieve a "perfect fit", so the effect of error re-ordering depends on the length of the time series. You discuss this briefly in the manuscript, but I think that this is among the most important mechanisms of BEAR, so it is worth demonstrating in isolation. For short time series (Figure 1), error re-ordering can already induce some degree of improvements by causing the marginal sample mean to follow the "ideal" error realizations; at the same time, the marginal error standard deviation decreases. This effect is exacerbated for longer time series (T=100, Figure 2, and T=1000, Figure 3). The consequence seems to be that the unordered, raw error distribution is "molded" to the ideal error realizations. I suspect (and you seem to share these suspicions in your responses) that in the limit of an infinitely long time series, error realizations would be compensated perfectly. This is important to discuss for prospective users of your manuscript, as it affects the algorithm's behaviour in somewhat unexpected ways.

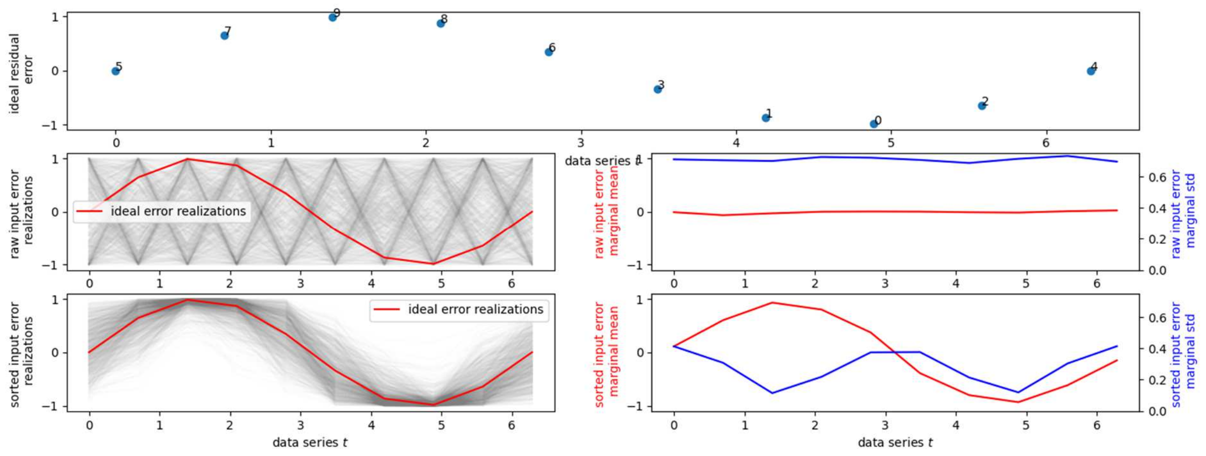

*Figure 1. Effect of error re-ordering for a perfect error distribution, an ensemble size of N=1000 and a time series length of T=10. The upper subplot shows the "ideal" realizations to compensate some residual error. The left centre plot shows N unordered realizations of an error distribution with the correct statistical moments of the ideal realizations (obtained by forming a cdf for a full sine wave). The right centre plot shows the marginal mean and standard deviation at each time step. The left bottom plot shows the N error realizations in the subplot above after ordering, and the right bottom plot shows the corresponding marginal mean and standard deviation.*

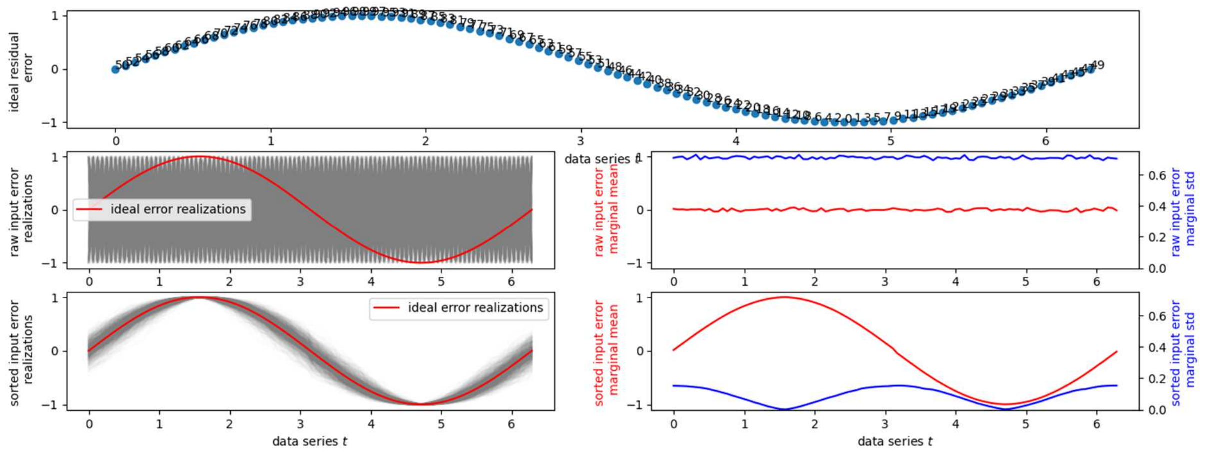
Figure 2. Effect of error re-ordering for a perfect error distribution, an ensemble size of N=1000 and a time series length of T=100. The upper subplot shows the “ideal” realizations to compensate some residual error. The left centre plot shows N unordered realizations of an error distribution with the correct statistical moments of the ideal realizations (obtained by forming a cdf for a full sine wave). The right centre plot shows the marginal mean and standard deviation at each time step. The left bottom plot shows the N error realizations in the subplot above after ordering, and the right bottom plot shows the corresponding marginal mean and standard deviation.

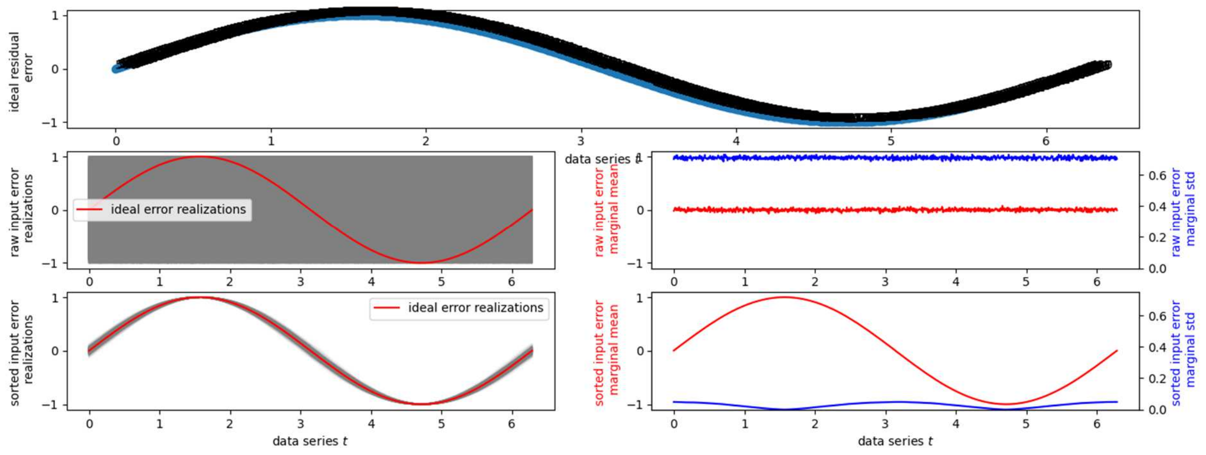
Figure 3. Effect of error re-ordering for a perfect error distribution, an ensemble size of N=1000 and a time series length of T=1000. The upper subplot shows the “ideal” realizations to compensate some residual error. The left centre plot shows N unordered realizations of an error distribution with the correct statistical moments of the ideal realizations (obtained by forming a cdf for a full sine wave). The right centre plot shows the marginal mean and standard deviation at each time step. The left bottom plot shows the N error realizations in the subplot above after ordering, and the right bottom plot shows the corresponding marginal mean and standard deviation.

Other interesting things to visualize might be what happens if the error distribution is not perfect (for this, just replace “vals = dist(np.random.uniform(size=(1000,resolution)))” in the code with some other distribution). You already show this indirectly in your models, but demonstrating this effect in isolation rather than through the lens of performance metrics might be a lot clearer. Feel free to take inspiration from my example code or use it directly. I have attached it at the end of this manuscript.

Of course, this code snippet just demonstrates what happens when we are re-ordering error realizations, not how you arrive at the error ranks and their interaction with the parameter inference in the first place (which are potentially additional topics to discuss). I hope that even if you decide to follow option (1), this snippet might give you some ideas on where to start with the Bayesian justification. Good luck!

**Specific comments** (any line numbers I list correspond to the non-track-changes manuscript):

Line 47: the same multiplier applied to one storm event

Maybe “the same multiplier applied to all time steps of one storm event” might be better, if I understand this part correctly; using “same” for a singular object (“storm event”) sounds a bit strange.

Line 184: the time scale is typically set as daily and the spatial scale is set as the catchment

This is not particularly clear. I assume you simulate in daily timesteps, and aggregate the catchment’s (presumably) surface area into a single spatial unit? If so, it might be better to replace this with “thus, we use daily time steps and consider the catchment a single, homogeneous spatial unit” or something along these lines.

Lines 302: From this point of view, it is more efficient to estimate the error rank than estimate the error value,

This sentence ends on a comma, not a period.

Line 309-311: Besides, to avoid the high-dimension calculation, modifying each input error according to its corresponding residual error only works in the rank domain. In the value domain, if there is no constraint on the estimated input errors, they will fully compensate for the residual error to maximize the likelihood function and subsequently be overfitted

This requires more discussion in the revised manuscript, as it seemingly contradicts what you write in the paragraph immediately prior: In the previous paragraph, you recommend sampling error realizations repeatedly and selecting the optimal realization to overcome “sampling bias” and improve the fit to the actual observations. However, in this paragraph you praise this very same sampling bias for preventing overfit. These are contradictory messages: provided you get the ranks right, if you were to resample an infinite number of times, you would eventually get an error realization which compensates the true error perfectly (even if your input error distribution is a really poor approximation to the “true” error distribution), thus negating your protection against overfit. The fact that this protection against overfit depends on the length of the time series might not be that much of an issue if you interpret your approach merely as a heuristic, but even in this case you need some practical guidelines on when to re-sample for short time series. The proposed dedicated section might help clearing some of this confusion up.

Line 320-322: Thus, unlike formal Bayesian inference, the BEAR method does not update the posterior distribution of the input errors, but identifies the input error through the deterministic relationship between the input error and model parameter.

As far as I can see, this is the first time you mention that BEAR is not a formal Bayesian inference method, so I suspect you would go for option (2). In any case, mentioning this in the discussion for the first time is a bit late: Something this important should be stated earlier, as early as the introduction or abstract. If you add the section exploring the consequences of error reordering, this would also be a good place to elaborate on this.

Aside from this manuscript structuring argument, the statement in these lines is also unfortunately wrong. Refer to the attached figures for demonstration. I also elaborate more on this two comments below.

Line 383: "However, the work in this study still identifies a few areas needing to be explored."

Nit-pick: This sentence is a bit unwieldy, in my opinion. How about "However, this study identifies a few areas which still need to be explored:"?

Response file / Response to major comment #7: "*A point of clarification: in each subsequent iteration of the BEAR algorithm, a new population of input errors are sampled from their a priori distribution. This means that the distribution of errors at each iteration is the same prior to reordering, i.e. the population of errors do not converge to a distribution that has different statistical features (mean, standard deviation, skewness).*"

You are of course correct, but I fear your comment misses the point somewhat: While it is undoubtedly true that the raw input error distribution never changes, BEAR *does* effectively update the input error distribution. In the figures from my code snippet, this can be seen in the bottom right subplots when compared to the centre right subplots. It becomes evident that the both the effective mean and standard deviation (and likely higher statistical moments as well) change dramatically after re-ordering. In essence, the fact that you are re-ordering transforms your raw error distribution and causes you to sample some different latent distribution instead. What this distribution really is remains the key question of your entire approach. If you can figure that out, you'll be one large step closer to justifying this approach theoretically. =)

In a sense, what you are doing seems distantly related to ideas in measure transport, see for example Marzouk et al. (2016) for an overview. In measure transport, the ultimate goal is to indirectly sample from an (almost) arbitrary target distribution. This is achieved by sampling a simple reference distribution instead (for example a multivariate standard Gaussian), then converting these reference samples through a deterministic function into samples from the target distribution. Of course, finding the correct transformation function is the key objective of this entire endeavour, and consequently its main challenge. In your study, you approach this from the opposite direction: you have some transformation, now you should find out what distribution you are sampling.

In summary, I would say the parallels to your approach are as follows: even though the reference distribution (corresponding to your raw input error distribution) never changes, the *pushforward distribution* (corresponding to the latent distribution your re-ordered error realizations are effectively sampled from) changes with the transformation function (in your case, the re-ordering according to different error ranks).

Marzouk, Y., Moselhy, T., Parno, M., & Spantini, A. (2016). An introduction to sampling via measure transport. *arXiv preprint arXiv:1602.05023*; https://arxiv.org/abs/1602.05023.

**Example code**.

```
import numpy as np

import matplotlib.pyplot as plt

from scipy.interpolate import interp1d

from matplotlib.gridspec import GridSpec

import copy


# Set a random seed

np.random.seed(0)


# Create some dummy error values to generate an artificial ranking

resolution  = 10 # Adjust this for different time series lengths


# Create a sequence of "ideal" error realizations following a sine curve

x = np.linspace(0,2*np.pi,resolution)

y = np.sin(x)


# Get the sorted indices of these error realizations

ranks   = np.argsort(y)


# Create a sorter list which can be used to reshape a sorted list according to

# the ranks we just defined

sorter  = np.argsort(ranks)


# Get the statistical moments of this residual error distribution

mean    = np.mean(y)

std     = np.std(y)


# Create a figure, plot the ranks for each error realization

plt.figure(figsize=(16,6))
```

```
gs = GridSpec(nrows=3,ncols=2)
plt.subplot(gs[0,:])
plt.scatter(x,y)
plt.xlabel('data series $t$')
plt.ylabel('ideal residual \n error')
for idx,i in enumerate(ranks):
    plt.text(x[i],y[i],str(idx))

# Create a cdf for a sine curve ----------------------------------------------

# Sample the sin cure with high resolution
dummy_sine_samples = np.sin(np.linspace(0,2*np.pi,10000))

# Sort these samples
dummy_sine_samples = np.sort(dummy_sine_samples)

# Search for potential duplicates
sineval,count = np.unique(dummy_sine_samples,return_counts=True)

# Create a cumulative sum for the counts
cumsum = np.cumsum(count,dtype=float)

# Standardize the cumulative sum
cumsum -= cumsum[0]
cumsum /= cumsum[-1]

# Create an interpolation function which takes a quantile as input and returns
# the corresponding sine value
dist    = interp1d(
    cumsum,
    sineval )
```

```
# Statistical moments of the raw input error distribution ---------------------

# Draw a thousand realizations of our custom error distribution
vals    = dist(np.random.uniform(size=(1000,resolution)))

# Plot the error realizations
plt.subplot(gs[1,0])
for n in range(1000):
    plt.plot(x,vals[n,:],color='grey',alpha=0.01)
plt.xlabel('data points $t$')
plt.ylabel('raw input error \n realizations')

# Plot the sine curve for reference
plt.plot(x,y,'r',label = 'ideal error realizations')
plt.legend()

# Calculate the mean and covariance of the samples
mean_1  = np.mean(vals,axis=0)
cov_1   = np.cov(vals.T)

# Plot the mean
plt.subplot(gs[1,1])
plt.plot(x,mean_1,'r')
plt.ylabel('raw input error \n marginal mean',fontdict={'color':'r'})
plt.ylim([-1.1,1.1])
ax2 = plt.gca().twinx()
ax2.plot(x,np.sqrt(np.diag(cov_1)),'b')
ax2.set_ylabel('raw input error \n marginal std',fontdict={'color':'b'})
ax2.set_ylim([0,0.75])
```

```
# Statistical moments of the sorted input error distribution ------------------

# Now sort the error realizations
vals_sorted = copy.copy(vals)
vals_sorted = np.sort(vals_sorted,axis=1)

# And redistribute them according to the ranks we determined
for n in range(vals.shape[0]):
    vals_sorted[n,:]   = vals_sorted[n,sorter]

# Plot the re-ordered error realizations
plt.subplot(gs[2,0])
for n in range(1000):
    plt.plot(x,vals_sorted[n,:],color='grey',alpha=0.01)
plt.xlabel('data series $t$')
plt.ylabel('sorted input error \n realizations')

# Plot the sine curve for reference
plt.plot(x,y,'r',label = 'ideal error realizations')
plt.legend()

# Calculate the statistical moments
mean_2  = np.mean(vals_sorted,axis=0)
cov_2   = np.cov(vals_sorted.T)

# Plot the mean
plt.subplot(gs[2,1])
plt.plot(x,mean_2,'r')
plt.ylabel('sorted input error \n marginal mean',fontdict={'color':'r'})
plt.ylim([-1.1,1.1])
plt.xlabel('data series $t$')
```

```
ax2 = plt.gca().twinx()

ax2.plot(x,np.sqrt(np.diag(cov_2)),'b')

ax2.set_ylabel('sorted input error \n marginal std',fontdict={'color':'b'})

ax2.set_ylim([0,0.75])


# Save the figure

plt.savefig('consequences_or_error_reordering_resolution_'+str(resolution)+'.png',bbox_inches='tig
ht')
```