ARL-TR-8828 • SEP 2019

Quantifying Uncertainties in Parameterizations of Strength Models of Rolled Homogeneous Armor: Part 3, Python-Based Workflow

by JJ Ramsey

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8828 ● SEP 2019

Quantifying Uncertainties in Parameterizations of Strength Models of Rolled Homogeneous Armor: Part 3, Python-Based Workflow

by JJ Ramsey
Computational and Information Sciences Directorate, CCDC Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
September 2019	Technical Report	October 2017-September 2019

4. TITLE AND SUBTITLE
Quantifying Uncertainties in Parameterizations of Strength Models of Rolled Homogeneous Armor: Part 3, Python-Based Workflow

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
JJ Ramsey

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Combat Capabilities Development Command Army Research Laboratory
ATTN: FCDD-RLC-NB
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8828

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes a workflow, based on the Python language and the Bayesian software tools PyStan and PyMC3, that has been used to obtain information on strength model parameters in rolled homogeneous armor that can be used in uncertainty propagation analyses. This workflow is supplemented with an illustration of how an approximate interval predictor model can be implemented using the Python software package SciPy. It is hoped that this workflow may serve as a source of example code for other CCDC Army Research Laboratory researchers who wish to obtain results that facilitate uncertainty quantification.

15. SUBJECT TERMS
uncertainty quantification, Bayesian analysis, Johnson-Cook, Zerilli-Armstrong, PyStan, PyMC3, interval predictor model

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James J Ramsey
Unclassified	Unclassified	Unclassified	UU	184	19b. TELEPHONE NUMBER (Include area code) 410-278-5614

Contents

List of Figures

List of Tables

1. Introduction

This report describes in detail a workflow for Bayesian analysis that uses the Python language[1] and two Bayesian software tools that work with that language, PyStan[2] and PyMC3.[3] There are two intended audiences for this report. One audience is the set of readers who have read the companion report[4] and wish to know further details of how to implement the analyses discussed within it. The other audience may not have read that report, but is still somewhat familiar with the broad strokes of Bayesian analysis and is looking for examples on how to implement it on something more than a "toy" example. For those in this second audience (as well as those in the first who need their memories refreshed), a few things are noted.

First, strength models for rolled homogeneous armor (RHA) are fit to the stress-strain data described in Appendix A, which come from the Material Implementation, Database, and Analysis Source (MIDAS). These data consist of n_c subsets, where subset i_c ($i_c \in [1, N_{i_c}]$) is associated with plastic strain rate $\dot{\epsilon}_p^{i_c}$ and temperature $T_{init}^{i_c}$, which is the initial temperature of an *unstrained* experimental sample. Each subset corresponds to one of the stress-strain curves shown in Fig. 1.

Second, the temperature rise during high-strain-rate deformation is approximately taken into account through the following equations:

$$T_j^{i_c} - T_{j-1}^{i_c} \approx \frac{\beta_{TQ}}{\rho c(T_{j-1}^{i_c})} \int_{\epsilon_{p,j-1}^{i_c}}^{\epsilon_{p,j}^{i_c}} \sigma d\epsilon_p \tag{1}$$

$$T_1^{i_c} - T_{init}^{i_c} \approx \frac{\beta_{TQ}}{\rho c(T_{init}^{i_c})} f_{area} \sigma_1^{i_c} \epsilon_{p,1}^{i_c}, f_{area} \in [0.5, 1] \tag{2}$$

Here, $T_{j-1}^{i_c}$, $\epsilon_{p,j}^{i_c}$, and $\sigma_j^{i_c}$ are, respectively, the temperature, plastic strain, and flow stress of data point j in subset i_c; β_{TQ} is the Taylor-Quinney coefficient; ρ is the density; and $c(T)$ is the specific heat, which is a function of temperature T. The integral in Eq. 1 is the area under the portion of stress-strain curve i_c that is over the strain interval $[\epsilon_{p,j-1}^{i_c}, \epsilon_{p,j}^{i_c}]$. The density is taken to be 7840 kg/m^3, following Benck.[5] The specific heat values for body-centered cubic (BCC) iron, in Appendix A, are assumed to approximate the specific heat values of RHA. The parameter f_{area} takes into account that when $\epsilon_{p,1}^{i_c} \neq 0$, $T_1^{i_c} \neq T_{init}^{i_c}$. While β_{TQ} is often taken to be equal to 0.9 for metals, there is a wide spread of values found in the literature, with β_{TQ} sometimes found to be as low as 0.4.[6] Estimation of f_{area} amounts to educated

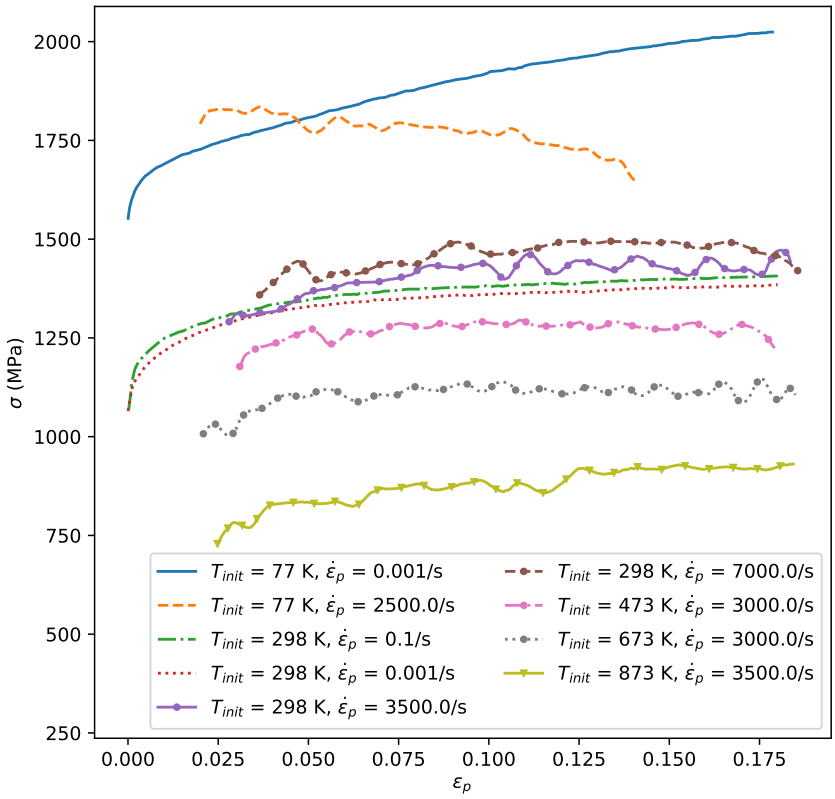


Fig. 1 Plots of flow stress σ vs. plastic strain ϵ_p for RHA from MIDAS, with the plastic strain rate denoted as $\dot{\epsilon}_p$ and the initial sample temperature T_{init}

guesswork. Accordingly, temperatures are estimated for a few combinations of reasonable estimates of β_{TQ} and f_{area}, shown in Table 1.

Table 1 Possible combinations of values of β_{TQ} and f_{area} used in temperature estimation

β_{TQ}	f_{area}
0.9	0.75
0.9	0.55
0.6	0.55
0.9	0.95
0.6	0.95

Third, the strength models to be fit are the Johnson-Cook model[7] and the Zerilli-Armstrong model for BCC materials.[8] These two models take the following forms,

$$\sigma_{JC}(\epsilon_p, \dot{\epsilon}_p, T^*; \boldsymbol{\theta}_{JC}) = (A + B\epsilon_p^n)[1 + C\ln(\dot{\epsilon}_p/\dot{\epsilon}_{p0})][1 - (T^*)^m] \tag{3}$$

$$T^* = (T - T_{room})/(T_{melt} - T_{room}) \tag{4}$$

$$\sigma_{ZA,BCC}(\epsilon_p, \dot{\epsilon}_p, T; \boldsymbol{\theta}_{ZA,BCC}) = C_0 + C_1 \exp[(-C_3 + C_4 \ln(\dot{\epsilon}_p/\dot{\epsilon}_{p0}))T] + C_5\epsilon_p^n \tag{5}$$

where σ_{JC} is the flow stress according to the Johnson-Cook model; $\sigma_{ZA,BCC}$ is the flow stress according to the Zerilli-Armstrong (BCC) model; ϵ_p is the plastic strain; $\dot{\epsilon}_p$ is the plastic strain rate; $\dot{\epsilon}_{p0} = 1/\text{s}$; T is the temperature; T_{room} is the room temperature; T_{melt} is the melting temperature; A, B, n, C, and m are fitting parameters of the Johnson-Cook model; $\boldsymbol{\theta}_{JC} = (A, B, n, C, m)$; C_0, C_1, C_3, C_4, C_5, and n are fitting parameters of the Zerilli-Armstrong (BCC) model; and $\boldsymbol{\theta}_{ZA,BCC} = (C_0, C_1, C_3, C_4, C_5, n)$. (There is no parameter C_2; such a parameter belongs to the face-centered cubic version of the Zerilli-Armstrong model.[9])

Fourth, because the experimental data for low strain rates come from a different measurement source than those for a high strain rate, the errors associated with each of them are different. The errors from both are assumed to be normally distributed, but the standard deviation of the noise from the low-strain-rate source is taken to be $SD_{\sigma,1}$, while that from the high-strain-rate source is taken to be $SD_{\sigma,2}$. These two standard deviations are taken to be nuisance parameters whose values are determined as part of Bayesian analysis.

Fifth, this report contains a workflow for sampling the *posterior predictive distribution*[10] (PPD) and the *pushed forward posterior*[11] (PFP), which can be used to check how well a model's predictions agree with the data. For the Bayesian models con-

sidered in this report, a sample from the PPD associated with experimental inputs $\epsilon_{p,j}^{i_c}$, $\dot{\epsilon}_p^{i_c}$, and $T_j^{i_c}$, $\sigma_j^{i_c,pred}(\epsilon_{p,j}^{i_c}, \dot{\epsilon}_p^{i_c}, T_j^{i_c})$, may be obtained as follows:

$$\begin{aligned} \sigma_j^{i_c,pred}(\epsilon_{p,j}^{i_c}, \dot{\epsilon}_p^{i_c}, T_j^{i_c}) &\sim \text{normal}(\sigma_{mdl}(\epsilon_{p,j}^{i_c}, \dot{\epsilon}_p^{i_c}, T_j^{i_c}; \boldsymbol{\theta}_{mdl}), SD_{\sigma,k}) \\ \{\boldsymbol{\theta}_{mdl}, SD_{\sigma,k}\} &\sim \mathcal{D}_{post} \end{aligned} \tag{6}$$

Here, $\mathcal{D}_{post}$ is the posterior distribution in Bayesian analysis, $k = 1$ for strain rates of 1/s or less, and $k = 2$ otherwise. Subscript "mdl" stands in for "JC" or "ZA, BCC." This sampling statement implies that a sample from the PPD is obtained by sampling $\boldsymbol{\theta}_{mdl}$ and $SD_{\sigma,k}$ from the posterior distribution, substituting that sample into the likelihood distribution (i.e., normal$(\sigma_{mdl}(\dots), SD_{\sigma,k})$) and then sampling from that likelihood. A sample of the PFP of the Bayesian models considered in this report may be obtained as follows:

$$\sigma_j^{i_c,pfp}(\epsilon_{p,j}^{i_c}, \dot{\epsilon}_p^{i_c}, T_j^{i_c}) \sim \sigma_{mdl}(\epsilon_{p,j}^{i_c}, \dot{\epsilon}_p^{i_c}, T_j^{i_c}; \boldsymbol{\theta}_{mdl}), \quad \text{if } \boldsymbol{\theta}_{mdl} \sim \mathcal{D}_{post} \tag{7}$$

This sampling statement implies that a sample from the PFP is obtained by sampling $\boldsymbol{\theta}_{mdl}$ from the posterior distribution and substituting that sample into the predictive model $\sigma_{mdl}(\dots)$.

Sixth, the Bayesian tools in this report implement Markov Chain Monte Carlo (MCMC), which produces one or more chains of samples from the posterior distribution in Bayesian analysis.[10,12] The particular MCMC algorithm used is Hamiltonian Monte Carlo[13] with the no-U-turn sampler (NUTS).[14]

Finally, an alternative approach, based on an interval predictor model (IPM),[15,16] is used to estimate the parameter uncertainty. An IPM is simply a function that returns an interval as its output rather than a single value. For example, given a function to predict the flow stress, $\sigma_{mdl}(\mathbf{e}, \boldsymbol{\theta}_{mdl})$ (where $\mathbf{e} \equiv (\epsilon_p, \dot{\epsilon}_p, T)$), and a set $\boldsymbol{\Theta}$, the interval within which the flow stress is estimated to lie is $[\sigma_{min}(\mathbf{e}; \boldsymbol{\Theta}), \sigma_{max}(\mathbf{e}; \boldsymbol{\Theta})]$, where

$$\sigma_{min}(\mathbf{e}; \boldsymbol{\Theta}) = \min_{\boldsymbol{\theta}_{mdl} \in \boldsymbol{\Theta}} \sigma_{mdl}(\mathbf{e}, \boldsymbol{\theta}_{mdl}) \tag{8}$$

$$\sigma_{max}(\mathbf{e}; \boldsymbol{\Theta}) = \max_{\boldsymbol{\theta}_{mdl} \in \boldsymbol{\Theta}} \sigma_{mdl}(\mathbf{e}, \boldsymbol{\theta}_{mdl}) \tag{9}$$

The set $\boldsymbol{\Theta}$ is chosen so as to keep the intervals from the IPM reasonably tight, given

known data points $\{\mathbf{e}_j^{i_c}, \sigma_j^{i_c}\}$. For example, $\mathbf{\Theta}$ may be chosen such that

$$\mathbf{\Theta} = \underset{\mathbf{\Theta}'}{\arg\min} \sum_{i_c=1}^{n_c} \sum_{j=1}^{N_{i_c}} \left[\sigma_{max}(\mathbf{e}_j^{i_c}; \mathbf{\Theta}') - \sigma_{min}(\mathbf{e}_j^{i_c}; \mathbf{\Theta}') \right] \tag{10}$$

The minimization of Eq. 10 under the constraint

$$\sigma_{min}(\mathbf{e}_j^{i_c}; \mathbf{\Theta}) \leq \sigma_j^{i_c} \leq \sigma_{max}(\mathbf{e}_j^{i_c}; \mathbf{\Theta}), \forall i_c \in [1, n_c], j \in [1, N_{i_c}] \tag{11}$$

may not be tractable, especially if there is no analytical solution to Eqs. 8 and 9, thus requiring a nested optimization (i.e., at each iteration to solve Eq. 10, optimization routines would need to be used to estimate σ_{min} and σ_{max} for each data point). However, one may obtain a more tractable problem by approximating $\sigma_{mdl}(\mathbf{e}, \mathbf{\theta}_{mdl})$ with a first-order Taylor expansion about a point estimate of $\mathbf{\theta}_{mdl}$, $\mathbf{\theta}_0$, and taking $\mathbf{\Theta}$ to be a hyperrectangle with corners $\mathbf{\theta}_0 - \Delta\mathbf{\theta}_{min}$ and $\mathbf{\theta}_0 + \Delta\mathbf{\theta}_{max}$. If $\mathbf{g}_{\sigma_{mdl}}(\mathbf{e})$ is the gradient of $\sigma_{mdl}(\dots)$ with respect to $\mathbf{\theta}_{mdl}$ evaluated at $\mathbf{e}$ and $\mathbf{\theta}_0$, and $|\mathbf{g}_{\sigma_{mdl}}(\mathbf{e})|$ is the *elementwise* absolute value of $\mathbf{g}_{\sigma_{mdl}}(\mathbf{e})$, then Eqs. 8 and 9 can be approximated as follows:

$$\begin{aligned} \sigma_{min}(\mathbf{e}; \mathbf{\Theta}) \approx \sigma_{mdl}(\mathbf{e}, \mathbf{\theta}_0) &- \frac{1}{2} \left(\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}) + |\mathbf{g}_{\sigma_{mdl}}(\mathbf{e})| \right)^T \Delta\mathbf{\theta}_{min} \\ &+ \frac{1}{2} \left(\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}) - |\mathbf{g}_{\sigma_{mdl}}(\mathbf{e})| \right)^T \Delta\mathbf{\theta}_{max} \end{aligned} \tag{12}$$

$$\begin{aligned} \sigma_{max}(\mathbf{e}; \mathbf{\Theta}) \approx \sigma_{mdl}(\mathbf{e}, \mathbf{\theta}_0) &- \frac{1}{2} \left(\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}) - |\mathbf{g}_{\sigma_{mdl}}(\mathbf{e})| \right)^T \Delta\mathbf{\theta}_{min} \\ &+ \frac{1}{2} \left(\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}) + |\mathbf{g}_{\sigma_{mdl}}(\mathbf{e})| \right)^T \Delta\mathbf{\theta}_{max} \end{aligned} \tag{13}$$

Here, a superscript T indicates the transpose. Given Eqs. 12 and 13 along with a fixed $\mathbf{\theta}_0$, Eq. 10 becomes

$$\Delta\mathbf{\theta}_{min}, \Delta\mathbf{\theta}_{max} = \underset{\Delta\mathbf{\theta}'_{min}, \Delta\mathbf{\theta}'_{max}}{\arg\min} \left[\sum_{i_c=1}^{n_c} \sum_{j=1}^{N_{i_c}} |\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}_j^{i_c})| \right]^T (\Delta\mathbf{\theta}'_{min} + \Delta\mathbf{\theta}'_{max}) \tag{14}$$

Together, Eqs. 11–14 form a constrained minimization problem that can be solved through linear programming.

Because this report is aimed primarily at those who have had little exposure to Bayesian analysis, it is written mostly in a step-by-step tutorial style, with the

Python code needed for analysis shown explicitly. Readers unfamiliar with Python may wish to view Appendix B. Excerpts and variables from program code, as well as filenames, are written in a fixed-width font `like this`.

2. Obtaining Software Tools

There are a variety of ways to obtain PyStan and PyMC3,[17,18] but here, instructions are presented for how to obtain them via the freely available Anaconda distribution.[19] (It is presumed here that any permissions needed to install software on one's computer have already been obtained, and that one can configure any anti-malware tools on that computer so that they will not interfere with launching of MCMC chains in parallel, an issue that has been a problem for some Stan users.[20]) First, if the distribution has not been installed already (and on certain CCDC Army Research Laboratory [ARL] workstations and computing clusters it may already be installed), then one should follow the installation instructions for Anaconda available online.[21] Since the details of these instructions depend on one's computing platform, they are not discussed here. Once the distribution is installed, one can use the Anaconda Navigator GUI to install various software that one may need. When one first starts Navigator,[22] one sees a window that looks like the one in Fig. 2. One can then click on the "Environments" tab (shown circled in a dashed red line), and then one should see a window like the one in Fig. 3.

At this point, one can then create a so-called "environment", which, loosely speaking, may be described as a container of software that one can maintain without it interfering with other software on one's system. To create an environment, one should first click on the "Create" button (shown circled in a dashed red line). This should produce a dialog window that looks like the one in Fig. 4. By default, the check box next to "Python" in the dialog window is already checked, so one should only need to provide a name for the environment. In these instructions, the name of this environment is "Bayes".

Once the environment has been created, one can then install PyStan and/or PyMC3. One first goes to the "Environments" tab, clicks on "Bayes", and then selects "Not installed" from the drop-down list over the list of software packages. The Anaconda Navigator window should then look like Fig. 5. From here, one can search for packages and click the check boxes next to the package(s) one wishes to install. For example, to install PyMC3, one can search for "pymc" and click the check box

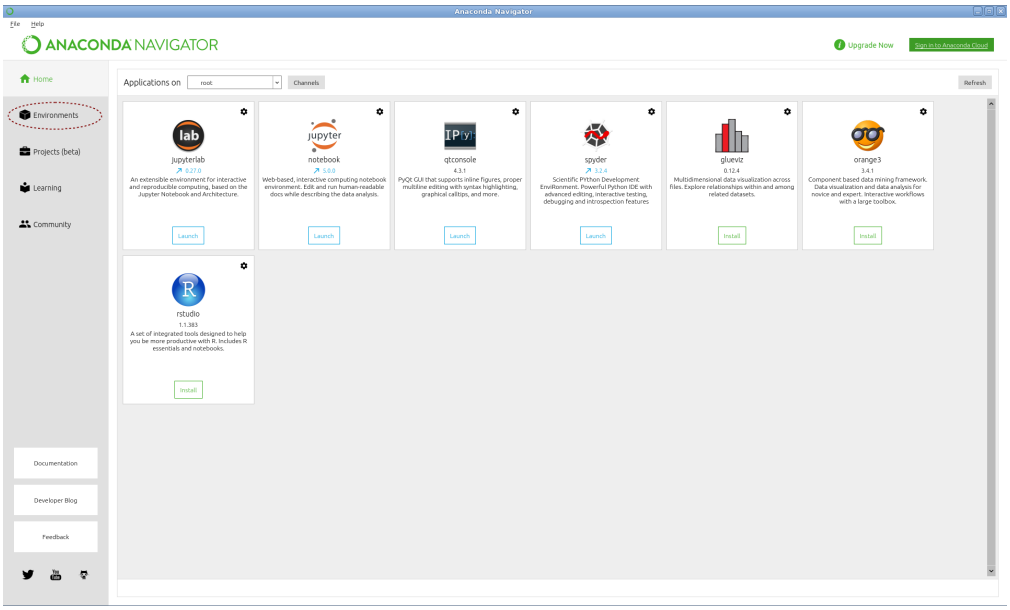


Fig. 2 Main window of Anaconda Navigator, with the "Home" tab shown and the "Environments" tab circled in a dashed red line

next to "pymc3" (*not* just "pymc"!), which looks like Fig. 6. (Unfortunately, during the writing of this report, a version of the PyMC3 package available from the Anaconda distribution, i.e., PyMC3 3.4.1, became broken.[23] A workaround for this is discussed later in this section.)

To finish installing, one clicks on the "Apply" button that appears at the lower right corner of the Navigator window. This also installs any packages on which PyStan and/or PyMC3 depend. To do the plotting and analysis in Python described in later sections, the packages Pandas and Matplotlib should be installed as well, if they have not already been installed as dependencies.

(If the available version of PyMC3 is still broken, one should still install it in order to install its dependencies, but then one should immediately uninstall it. One should then install PyMC3 from the Python Package Index [PyPI]. To do this, one should go to the "Environments" tab shown in Fig. 3, click on the green triangle next to "Bayes", choose "Open in Terminal" from the resulting pop-up menu, type `pip install pymc3` in the resulting terminal, and press Enter. When using Anaconda, though, installing from PyPI should be a last resort.)

If one clicks on the "Home" tab in Anaconda Navigator (the tab just above the

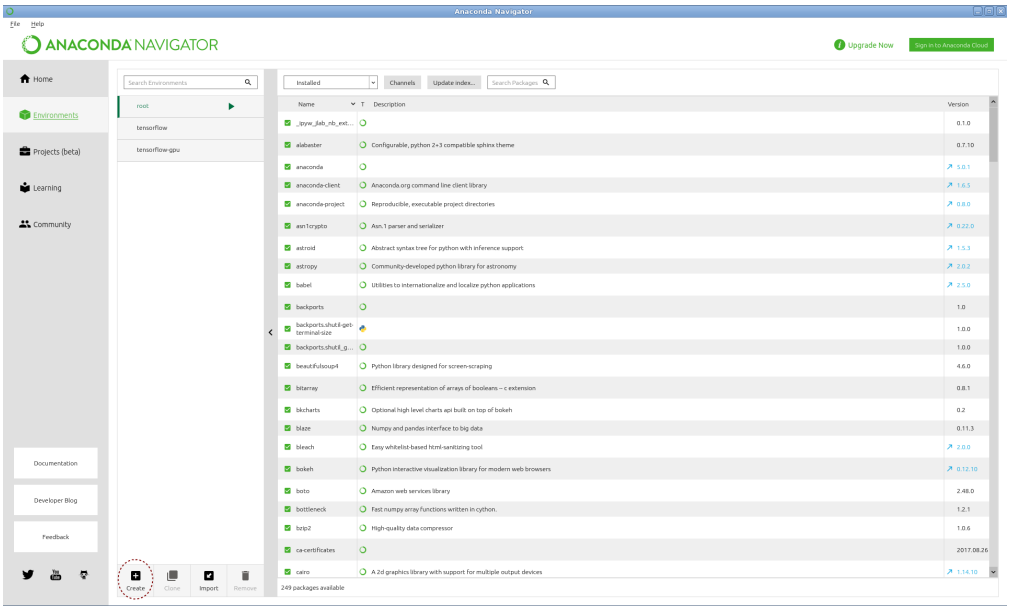


Fig. 3 Main window of Anaconda Navigator, with the "Environments" tab shown and the "Create" button circled in a dashed red line

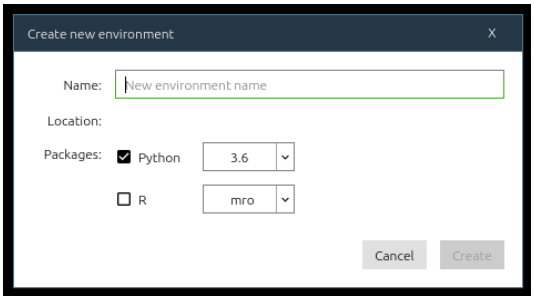


Fig. 4 Dialog window for creating environments with Anaconda Navigator, with default settings

"Environments" tab) and chooses the item "Bayes" from the drop-down menu next to the text "Applications on", one can see a window like the one in Fig. 7. From this window, one can launch Jupyter Notebook,[24] where one can write and execute Python code in an incremental, piecemeal fashion and intersperse the code with text explaining one's intended workflow. One can also install and then launch Spyder,[25] an integrated development environment for Python.

For those who use the text editors Emacs or Vim, syntax highlighting for the Stan language is available as well.[26,27]

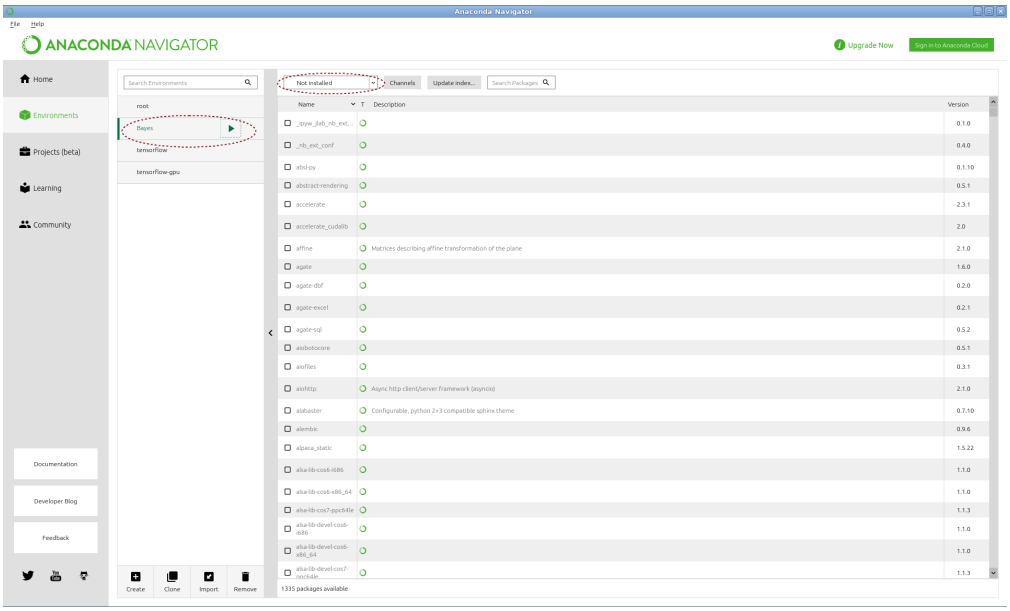


Fig. 5 Main window of Anaconda Navigator, with the "Environments" tab showing a list of available software packages that are not installed. The entry for the "Bayes" environment and the drop-down list with the entry "Not installed" are both circled in a dashed red line.

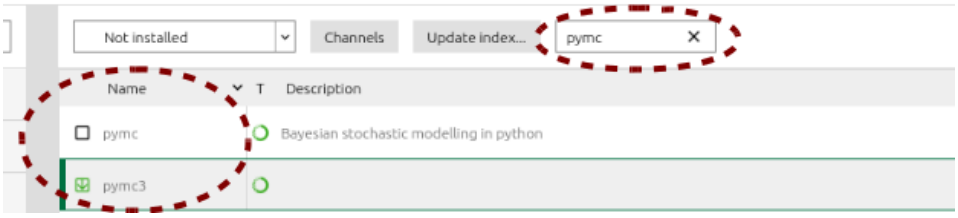


Fig. 6 Close-up of main window of Anaconda Navigator, with the "Environments" tab showing a list of available software packages with the string "pymc" in their names. The search box and package names are circled in a dashed red line.

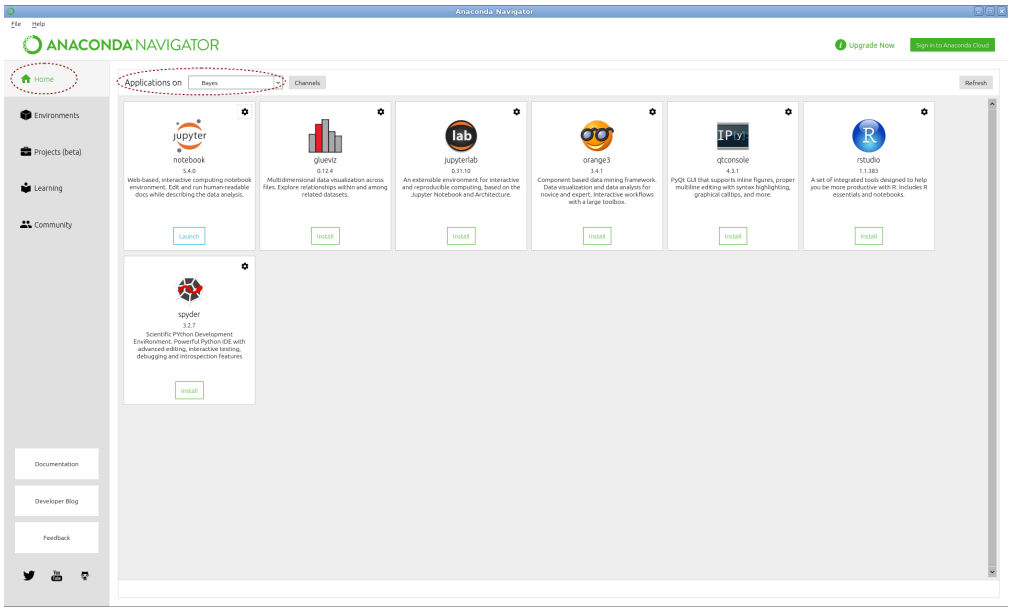


Fig. 7 Main window of Anaconda Navigator, with the "Home" tab shown and the "Environments" tab circled in a dashed red line. Applications for the "Bayes" environment are shown.

3. Working Directories

It is presumed that all code and data in the following analyses are in subdirectories immediately below some user-chosen base directory. Subdirectory `stan_model_specs` contains all Stan model specification files. Subdirectory `Python` is the working directory from which all Python code is executed, and any Python module files that are not part of a Python installation are in this directory as well. The subdirectory `MIDAS_data` contains the stress-strain data from MIDAS described in Appendix A, and the subdirectory `Other_data` contains other files that can be processed with multiple programming languages.

4. Data Files

The original data from MIDAS have been stored in a set of comma-separated value (CSV) files, with one file for a given strain rate and initial temperature. The data from these files are shown in Appendix A. For each file, the first column is the *plastic* strain (so no conversion from total strain to plastic strain is needed here), and the second column is the true stress in megapascals. There are no column headings in the CSV files. The naming convention for each file indicates the temperature and strain rate for which the data have been determined. For example, in the filename `T298K_edot0.1_per_s.csv`, "`T298K`" indicates that the initial temperature is 298 K, and "`edot0.1_per_s`" indicates that the strain rate is 0.1/s.

The `Other_data` directory mentioned in Section 3 contains a CSV file named `Austin_Specific_Heat_BCC_Iron.csv` that has the specific heat data as a function of temperature for BCC iron. The first column is the absolute temperature in kelvin, and the second column is the specific heat in $\mathrm{J}/(\mathrm{kg} \cdot \mathrm{K})$. The data from this file are also in Appendix A.

Also in the `Other_data` directory are JavaScript Object Notation (JSON) files used for the parameters for priors, as well as some other miscellaneous data. The reason for putting these parameters into files is that they are used repeatedly in both the process of fitting models and in later data analysis. The reason for using JSON files in particular is that they are human-readable text files that can easily be read into both R and Python sessions, and thus can be used not only in the workflow discussed in this report but in the R workflow discussed in Ramsey.[28]

Next are the contents of the data file `JC_priors.json`, which pertains to the

weakly informative priors of the Johnson-Cook model discussed in Ramsey.[4]

```
{
    "A_guess_mean" : 1000.0,
    "A_guess_sd" : 333.333333333333,

    "B_guess_mean" : 1000.0,
    "B_guess_sd" : 333.333333333333,

    "C_guess_mean" : 0.001,
    "C_guess_sd" : 0.000333333333333333,

    "m_guess_mean" : 1.0,
    "m_guess_sd" : 0.333333333333333,

    "n_alpha" : 1.1,
    "n_beta" : 1.1,

    "sd_sigma_guess_mean" : [100.0, 100.0],
    "sd_sigma_guess_sd" : [33.3333333333333, 33.3333333333333]
}
```

Between the curly braces is a comma-separated list of key-value pairs, where the keys are strings, the values are either numbers or lists of numbers in brackets, and a colon is used to separate the keys and values. Here, the keys correspond to data variables in the Stan specification file in Section D.1.

As discussed in Ramsey,[4] a strongly informative prior may also be used for parameter A of the Johnson-Cook model. The mean and standard deviation of this prior, based on experimental data from Benck,[5] are stored in a JSON file named `JC_prior_A_Benck.json`:

```
{
    "A_guess_mean": 707.25,
    "A_guess_sd": 10.63
}
```

There are also other quantities that are needed for the fit of the Johnson-Cook model, and since these quantities are also used later, they are saved to a JSON file, entitled `JC_other_data.json`:

```
{
    "T_room" : 298.0,
    "T_melt" : 1783.0,
    "epsilon_p_dot_0" : 1.0
}
```

This file, of course, has the values of parameters T_{melt}, T_{room}, and $\dot{\epsilon}_{p0}$.

Parameters for the priors of the Zerilli-Armstrong (BCC) model are in the file `ZA_BCC_priors.json`:

```
{
    "C0_guess_mean" : 100.0,
    "C0_guess_sd" : 33.3333333333333,

    "C1_guess_mean" : 1000.0,
    "C1_guess_sd" : 333.333333333333,

    "C3_guess_mean" : 1e-3,
    "C3_guess_sd" : 3.33333333333333e-04,

    "C4_guess_mean" : 1e-05,
    "C4_guess_sd" : 3.33333333333333e-06,

    "C5_guess_mean" : 1000.0,
    "C5_guess_sd" : 333.333333333333,

    "n_alpha" : 1.1,
    "n_beta" : 1.1,

    "sd_sigma_guess_mean" : [100.0, 100.0],
    "sd_sigma_guess_sd" : [33.3333333333333, 33.3333333333333]
}
```

The keys correspond to data variables in the Stan specification file in Section D.2. The values associated with these keys make the priors of the Zerilli-Armstrong (BCC) model weakly informative.

5. Testing Models with Simulated Data

A Bayesian model should be tested with simulated data, that is, data sampled from the likelihood of the model given known model parameters and other model inputs, which in this case are the strain, strain rate, and temperature. When one fits the model back to these simulated data, the resulting point estimates for the model parameters should be approximately the same as the parameter values that one used to create the simulated data in the first place. If not, that means that the model should be revised. Examples of this sort of testing are shown for both the Johnson-Cook and Zerilli-Armstrong (BCC) models.

5.1 Functions for Testing Models

Some custom Python functions, whose sources are in the file `bayes_stress_strain_utils.py` in Appendix C, have been used in the testing of models described later on. These functions are as follows:

- `simulate_data`, which is used to help generate the kind of simulated data described previously, while accounting for the temperature rise estimated in Eq. 1;
- `gen_lin_interp_func`, which is used to generate a function that linearly interpolates tabular data (in particular, the specific heat data in Appendix A);
- `plot_stress_strain_curves`, which creates a plot of several stress-strain curves and writes it to a file;
- `print_stan_summary`, which generates summary statistics and some diagnostics from the results of an MCMC run with PyStan;
- `read_from_json_file`, which reads a Python object from a (possibly Gzip-compressed[29]) JSON file;
- `read_from_pickle_file`, which reads a Python object from a (possibly Gzip-compressed[29]) Python pickle file; and
- `save_to_pickle_file`, which saves a Python object to a (possibly compressed) Python pickle file. It ensures that any directories in the file path supplied to it actually exist and creates them if they do not. If the file path ends in "`.gz`", then the resulting pickle file is Gzip-compressed.[29]

The details of these functions may be mainly of interest to readers who are looking for example code to use as a reference. However, two of these functions are discussed in more detail: `simulate_data`, whose contents pertain to the physics and mathematics of the sample problem, and `print_stan_summary`, which exists largely as a workaround for limitations of the current release of PyStan at the time of writing. Thus, these are discussed in more detail.

The function `simulate_data` has the following arguments:

- `sigma_model_func`, a function representing the strength model (e.g., σ_{JC} or $\sigma_{ZA,BCC}$ from Eqs. 3 and 5) that returns the flow stress and takes four arguments: plastic strain, plastic strain rate, temperature, and some data structure containing the model parameters (such as a Python dictionary)
- `epsilon_p_max`, the largest plastic strain for which stresses are calculated
- `epsilon_p_dot`, the plastic strain rate
- `T_init`, the initial temperature of the sample
- `theta_model` the model parameters of the strength model (e.g., $\boldsymbol{\theta}_{JC}$ or $\boldsymbol{\theta}_{ZA,BCC}$ from Eqs. 3 and 5)
- `beta_TQ`, the Taylor-Quinney coefficient
- `rho`, the density of the sample
- `specific_heat_func`, a function that returns the specific heat for a given temperature (which can and later on are generated using `gen_lin_interp_func`)
- `curve_size`, the number of data points in the stress-strain curve

The following statement creates a 1-D NumPy array of length `curve_size`, `epsilon_p`, that contains evenly spaced strain values from 0 to `epsilon_p_max`:

```
epsilon_p = np.linspace(0.0, epsilon_p_max, curve_size)
```

The next two statements create the 1-D NumPy arrays `T` and `sigma`, which are to hold sequences of length `curve_size` that contain temperatures and stresses, respectively. While the size of these arrays is set, their contents are uninitialized and contain random garbage at this point.

```
T = np.empty(curve_size)
sigma = np.empty(curve_size)
```

After this come the parts of the function that generate simulated temperature and stress data. There is a potential circularity here. In general, the temperature depends upon the stress, but to calculate the stress from the strength model, one needs the temperature. To work around this, the temperature in element i of `T` is estimated

using the stress in element $i - 1$ of `sigma`. To bootstrap this process, the first elements of `T` and `sigma` are set using the initial temperature `T_init`:

```
T[0] = T_init

sigma[0] = sigma_model_func(
    epsilon_p[0],
    epsilon_p_dot,
    T[0],
    theta_model
)
```

At this point, the rest of the elements of `T` and `sigma` can be set as follows:

```
for i in range(1, curve_size):
    # Estimate of area under stress-strain curve from
    # epsilon_p[i-1] to epsilon_p[i-1].
    area_under_curve = sigma[i-1]*(epsilon_p[i] - epsilon_p[i-1])

    T_rise = beta_TQ*area_under_curve/(
        rho*specific_heat_func(T[i-1]))

    T[i] = T[i-1] + T_rise

    sigma[i] = sigma_model_func(
        epsilon_p[i],
        epsilon_p_dot,
        T[i],
        theta_model
    )
```

As the comment in the previous Python code indicates, `area_under_curve` is an estimate of the area under the portion of the stress-strain curve that is over the interval [`epsilon_p[i - 1]`, `epsilon_p[i]`]. This corresponds to the integral in Eq. 1, with the integrand being approximated as a constant with the value `sigma[i - 1]`. The temperature rise `temp_rise` also follows from Eq. 1. Once the temperature rise is estimated, then it is straightforward to determine `T[i]` and then `sigma[i]`.

Finally, the function returns its values in a Python dictionary as follows:

```
return {"T" : T,
        "epsilon_p": epsilon_p,
        "sigma": sigma
}
```

The function `print_stan_summary` mainly exists to account for two issues

present in the release versions of PyStan available at the time of writing: a bug that may cause PyStan to print spurious not-a-number (NaN) values in one of its diagnostics, and certain diagnostic checks being missing in PyStan but present in RStan. Both of these issues have been fixed in the current development version of PyStan.[30,31] The function requires only one argument, `fit`, a PyStan `StanFit4model` object that contains the results of an MCMC run.[32] It then prints out a summary of statistics of that run, such as the mean and standard deviation of model parameters, as well as indicators of problems with the run.

The body of the function begins with a workaround for the aforementioned PyStan bug.

```
try:
    pystan.constants.EPSILON = float("-inf")
except:
    # If there is an exception, the constant is longer in
    # PyStan, so the workaround is not needed.
    pass
```

The constant `pystan.constants.EPSILON` is used in an attempt to avoid a divide-by-zero error. When a denominator used to calculate a diagnostic called "`Rhat`" is less than this constant, it is presumed to be effectively zero, and the diagnostic value is set to NaN. Unfortunately, this leads to small but still valid denominator values to be spuriously treated as zero. To fix this problem, `pystan.constants.EPSILON` is changed from its original value of 10^{-6} to $-\infty$. To be compatible with future versions of PyStan, in which the constant `pystan.constants.EPSILON` is not available, the setting of this constant is within a Python `try`/`except` control structure. If setting the constant fails, then nothing happens.

The following code prints some diagnostics and a summary:

```
# Check for divergent transitions
stan_utility.check_div(fit)

# Check if transitions hit maximum treedepth
stan_utility.check_treedepth(
    fit,
    max_depth = int(fit.stan_args[0]["ctrl"]["sampling"]["max_treedepth"])
)

# Check if BFMI is low
stan_utility.check_energy(fit)
```

```
# Print summary statistics
print(fit)
```

The diagnostic functions are from the `stan_utility` module written by Betancourt,[33] a copy of which is in the `Python` directory mentioned in Section 3. The meanings of these diagnostics are discussed by the Stan Development Team.[34]

5.2 Making Simulated Data for Johnson-Cook Model

Simulated stress-strain curves are to be created for several combinations of plastic strain rate and initial sample temperature, such as those in the following Python lists:

```
epsilon_p_dot = [0.001,   0.1, 3500.0, 7000.0, 3000.0, 3000.0]
T_init =        [298.0, 298.0,  298.0,  298.0,  473.0,  673.0]
```

The values in these lists are taken from Meyer and Kleponis.[35] They are in units of s^{-1} and kelvin, respectively. To account for the temperature rise during deformation of the sample, the sample density ρ and the specific heat as a function of temperature $c(T)$ are needed. Density ρ can be trivially represented by the Python variable `rho`:

```
rho = 7840.0 # kg/m^3
```

Representing the specific heat function in Python is less straightforward. As mentioned in Section 4, the specific heat data Appendix A has been collected into the CSV file `Austin_Specific_Heat_BCC_Iron.csv`. This can be read into a Python session as follows, using the Python module Pandas[36]:

```
import os
import pandas

c_data = pandas.read_table(
    os.path.join(os.pardir, "Other_data",
                 "Austin_Specific_Heat_BCC_Iron.csv"),
    sep = ",",
    header = None
)
```

The contents of the CSV file are stored in a data frame named `c_data`. The argument "`header = None`" indicates that the first line of the CSV file should not be treated as column headers. Because the simulated stress data are supposed to be in megapascals, the specific heat values need to be in compatible units. Accordingly, the second column of `c_data`, which contains these values, is modified as follows:

```
# Conversion factor from MPa to Pa
MPa_to_Pa = 1e6

c_data.iloc[:,1] /= MPa_to_Pa
```

After this modification is done, a function that estimates the specific heat as a function of temperature can be generated as follows, using functionality from SciPy[37]:

```
from scipy.interpolate import interp1d

c_func = interp1d(c_data.iloc[:,0],
                  c_data.iloc[:,1],
                  fill_value = "extrapolate")
```

The function `c_func` linearly interpolates the specific heat data from `c_data`. In the unlikely event that extrapolation from the data is needed, the argument "`fill_value = "extrapolate"`" allows for this. The function `gen_lin_interp_func` from the Python module `bayes_stress_strain_utils.py` in Appendix C encapsulates most of the previous steps used to obtain `c_func` and is used for obtaining such a function in later parts of this report.

To create simulated data for the Johnson-Cook model, one needs a Python function specifying this model. Here, the function `jc` from the module file `jc.py` in Appendix C is used for this purpose. However, to use this with the function `simulate_data` (also in Appendix C as part of `bayes_stress_strain_utils.py`), which needs a function that not only represents the strength model but has certain arguments in a certain order, a wrapper function needs to be used:

```
from jc import jc
import numpy as np

def sigma_model_func(epsilon_p,
                     epsilon_p_dot,
                     T,
                     theta_model):

    log_epsilon_p_dot = np.log(
        np.asarray(epsilon_p_dot)/theta_model["epsilon_p_dot_0"]
    )

    return jc(
        epsilon_p,
        log_epsilon_p_dot,
```

```
        (np.asarray(T) - theta_model["T_room"])/
          (theta_model["T_melt"] - theta_model["T_room"]),
        theta_model["A"],
        theta_model["B"],
        theta_model["n"],
        theta_model["C"],
        theta_model["m"]
    )
```

In the wrapper function, the NumPy function `asarray` is used to wrap the variables `epsilon_p_dot` and `T` so that they may be used in array arithmetic, and `np.log` is a function for the natural logarithm, The model parameters needed by `sigma_model_func` are shown:

```
theta_model = {
    "A": 780.0, # MPa
    "B": 780.0, # MPa
    "n": 0.106,
    "C": 0.004,
    "m": 1.0,
    "T_melt": 1783.0, # Kelvin
    "T_room": 298.0, # Kelvin
    "epsilon_p_dot_0": 1.0 # per s
}
```

The values of these parameters happen to be from Meyer and Kleponis,[35] but in principle, they could be set to any plausible values. At this point, nearly all the information needed for `simulate_data` has been input to a Python session. The maximum value for ϵ_p and values for $SD_{\sigma,1}$ and $SD_{\sigma,2}$ are set as follows:

```
epsilon_p_max = 0.2
sd_sigma = [1.0, 10.0]
```

Finally, the simulated data can be generated as follows. Here, `beta_TQ`, the Taylor-Quinney coefficient, is set to zero for low strain rates to simulate the lack of a temperature rise at those rates. Similarly, `curr_sd_sigma` is either $SD_{\sigma,1}$ or $SD_{\sigma,2}$, depending on the strain rate. For the sake of reproducibility, the following code sets the seed used by functions in the NumPy submodule `random`.

```
import bayes_stress_strain_utils as bssu

# Initializing to empty lists
sigma = []
epsilon_p = []
T = []

min_curve_size = 40
```

```
max_curve_size = 50

np.random.seed(12345)

for i in range(len(epsilon_p_dot)):

    if epsilon_p_dot[i] <= 1.0:
        beta_TQ = 0.0
        curr_sd_sigma = sd_sigma[0]
    else:
        beta_TQ = 0.9
        curr_sd_sigma = sd_sigma[1]

    # Sets curve_size to a random integer between min_curve_size
    # and max_curve_size
    curve_size = np.random.randint(min_curve_size,
                                   max_curve_size + 1)

    curr_data = bssu.simulate_data(
        sigma_model_func,
        epsilon_p_max,
        epsilon_p_dot[i],
        T_init[i],
        theta_model,
        beta_TQ,
        rho,
        c_func,
        curve_size
    )

    sigma.append(curr_data["sigma"] +
                 curr_sd_sigma*np.random.randn(curve_size))

    epsilon_p.append(curr_data["epsilon_p"])
    T.append(curr_data["T"])
```

Here, `sigma`, `epsilon_p`, and `T` are lists of 1-D NumPy arrays, and `sigma[i]` and `epsilon_p[i]` are the stresses and strains for stress-strain curve `i`. Furthermore, `T[i]` is an array of temperatures, such that `T[i][j]` is the temperature for data point `j` of stress-strain curve `i`. This code takes into account that if $\sigma \sim \text{normal}(\mu_\sigma, SD_\sigma)$, then $\sigma = \mu_\sigma + SD_\sigma e_{\text{normal}}$, where $e_{\text{normal}} \sim \text{normal}(0, 1)$. The call to the function `randn` from the NumPy submodule `random`) provides an array of length `curve_size` containing random values sampled from normal$(0, 1)$, while μ_σ here is `curr_data["sigma"]` and SD_σ here is `curr_sd_sigma`.

As a sanity check, one may plot the simulated data to a ".pdf" file (in the directory `plot_files`) with the function `plot_stress_strain_curves` (also from the module file `bayes_stress_strain_utils.py` in Appendix C). An

example usage of this function is shown:

```
bssu.plot_stress_strain_curves(
    os.path.join("plot_files", "jc_simulated_data.pdf"),
    epsilon_p_dot,
    T_init,
    epsilon_p, sigma,
    space_for_legend = 0.0
)
```

The resulting plot of the simulated data is in Fig. 8.

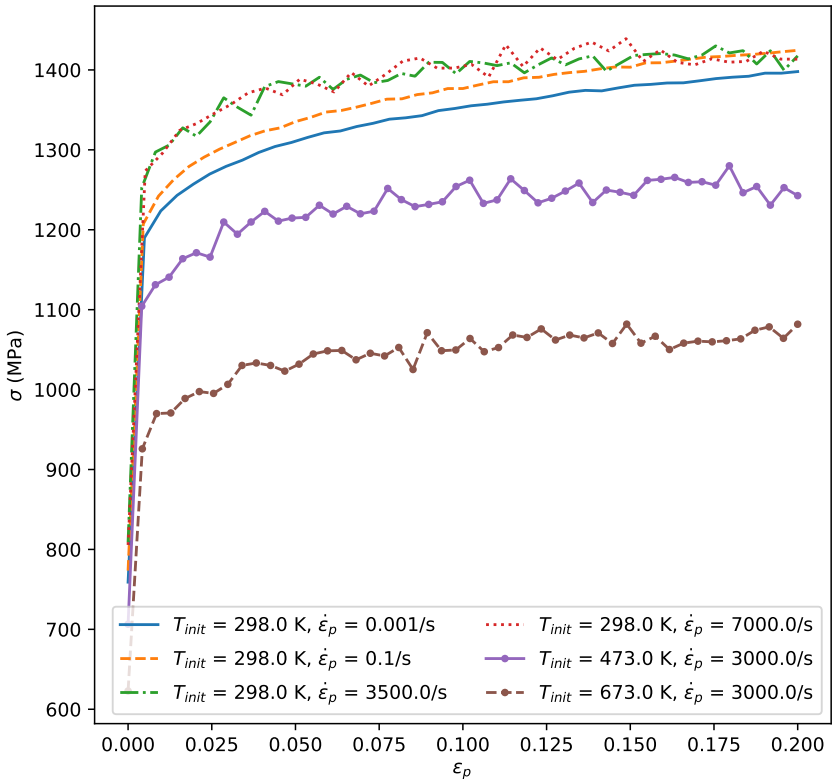


Fig. 8 Plot of simulated data used to test the Johnson-Cook PyStan and PyMC3 models

At this point, the simulated data can begin to be converted into forms more suitable for PyStan and PyMC3 and then saved to Gzip-compresssed[29] Python pickle files. Both of these need variables pertaining to the priors (that is, `A_guess_mean`, `n_alpha`, etc.) so these are set, using the JSON file `JC_priors.json` from

Section 4.

```
prior_params = bssu.read_from_json_file(
    os.path.join(os.pardir, "Other_data", "JC_priors.json")
)
```

With PyStan, input for MCMC sampling needs to be a dictionary whose keys are the variable names in the `data` block of a Stan specification file. Such a dictionary is created and saved to a Python pickle file in the directory `pkl_data_files` as shown:

```
dict_for_pystan = {
    "num_curves": len(epsilon_p_dot),
    "curve_sizes": [len(s) for s in sigma],
    "epsilon_p_dot": epsilon_p_dot,
    "epsilon_p": np.concatenate(epsilon_p),
    "sigma": np.concatenate(sigma),
    "T": np.concatenate(T),
    "T_melt": theta_model["T_melt"],
    "T_room": theta_model["T_room"],
    "epsilon_p_dot_0": theta_model["epsilon_p_dot_0"]
}

dict_for_pystan.update(prior_params)

bssu.save_to_pickle_file(
    dict_for_pystan,
    os.path.join("pkl_data_files",
                 "jc_simulated_data_pystan.pkl.gz")
)
```

Here, `np.concatenate` is used to concatenate all the arrays in the list `sigma` into one long 1-D array corresponding to the vector `sigma` in the specification file `jc.stan`. The same goes for `epsilon_p` and `T`. The dictionary key `"curve_sizes"` is associated with a list whose elements are the lengths of the arrays in `sigma`. This is done to accord with how data storage for stresses, strains, and temperatures is specified in the file `jc.stan`, as illustrated in Fig. 9, as a workaround for Stan's lack of support for ragged arrays.[38] To save space, the pickle files of the simulated data are Gzip-compressed.[29]

For PyMC3, the data to be saved are what are to be passed as arguments to `make_jc_model` in the module `jc_pymc3.py` (in Appendix C). These arguments are placed in a dictionary and saved as follows:

```
bssu.save_to_pickle_file({
        "epsilon_p_dot": epsilon_p_dot,
```

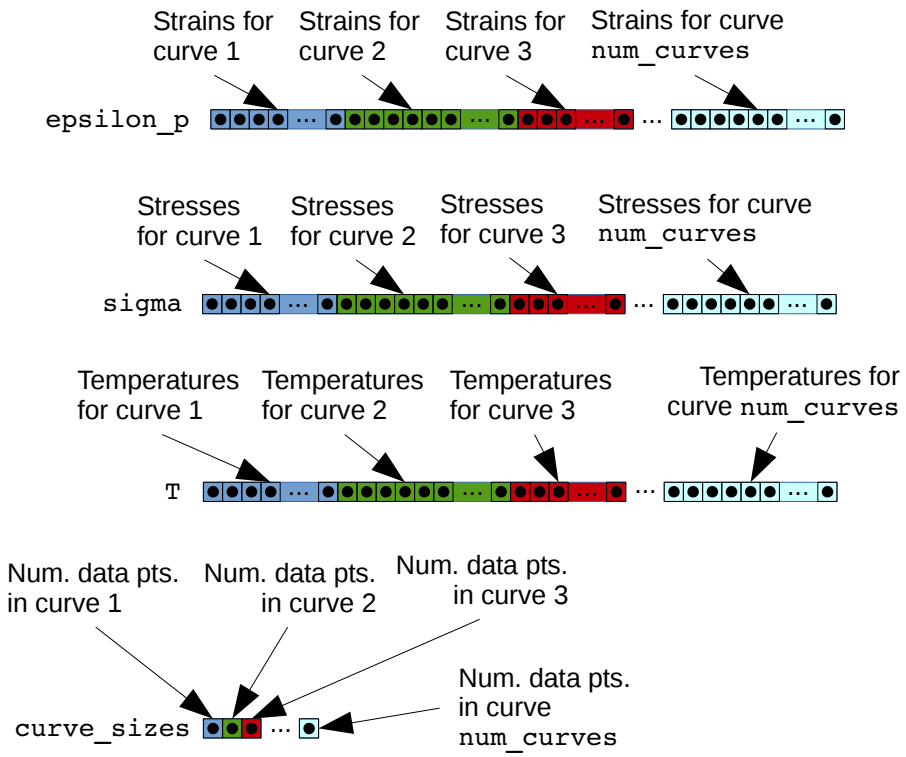


Fig. 9 Storage of data for stress-strain curves in the Stan vectors `epsilon_p`, `sigma`, `T`, and `curve_sizes`

```
        "epsilon_p": epsilon_p,
        "sigma": sigma,
        "T": T,
        "T_melt": theta_model["T_melt"],
        "T_room": theta_model["T_room"],
        "epsilon_p_dot_0": theta_model["epsilon_p_dot_0"],
        "prior_params": prior_params
    },
    os.path.join("pkl_data_files",
                 "jc_simulated_data_pymc3.pkl.gz")
)
```

5.3 Testing Johnson-Cook Model with PyStan

First, one should import the PyStan module (named `pystan`), if only to make sure it is actually there. This may simply be done with the following line of code:

```
import pystan
```

At this point, a Stan specification file for the Johnson-Cook model should have been written separately in a text editor. Here, the file is named `jc.stan`, and its contents are shown in Appendix D. As discussed in Section 3, it is in the directory `stan_model_specs`, which is a sibling to the directory `Python`, the working directory where Python code is being executed. Accordingly, the path to the Stan specification file `jc.stan` can be specified as follows:

```
import os
path_to_jc_stan_file = os.path.join(os.pardir,
                                    "stan_model_specs",
                                    "jc.stan")
```

The file `jc.stan` can be *compiled* into a `StanModel` object named `jc_model`:

```
jc_model = pystan.StanModel(path_to_jc_stan_file)
```

Even if the compilation has succeeded, there may still be warnings, especially from the underlying C++ compiler that PyStan uses to create `jc_model`. Most warnings, especially one about `auto_ptr` being deprecated, may be safely ignored, but out of caution it is best to at least read them.

To save the `StanModel` object for future use, one can save it to a Python pickle file. This file, named `jc.pkl`, is created and stored in the subdirectory `compiled_stan_models`, using the function `save_to_pickle_file` from the `bayes_stress_strain_utils` module used in Section 5.2:

```
import bayes_stress_strain_utils as bssu
bssu.save_to_pickle_file(jc_model,
                         os.path.join("compiled_stan_models",
                                      "jc.pkl"))
```

One can use the `read_from_pickle_file` function, also from the `bayes_stress_strain_utils` module, to bring the `StanModel` object stored in `jc.pkl` into another Python session. However, a pickle file of a `StanModel` object onlys work with Python sessions done on the same system used to generate the file, or at least a system that is nearly identical.

At this point, the simulated data may be loaded into the Python session, and the Johnson-Cook model may be loaded as well, if it has not been loaded already:

```
import bayes_stress_strain_utils as bssu
import os.path

my_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files", "jc_simulated_data_pystan.pkl.gz"))

jc_model = bssu.read_from_pickle_file(
    os.path.join("compiled_stan_models", "jc.pkl"))
```

One then fits the model to the simulated data via the `sampling` method of a `StanModel` object:

```
import time

start_time = time.perf_counter()
jc_fit = jc_model.sampling(data = my_data,
                           seed = 12345)
elapsed_time = time.perf_counter() - start_time
```

For the sake of reproducibility, the seed for random number generation is set via the `seed` argument of the `sampling` method. Also, since the only way PyStan prints elapsed time is via terminal output (which may not be visible in, for example, a Jupyter notebook), the functionality of Python's `time` module is used to estimate the elapsed time in seconds. One should note that the elapsed time may be affected by processes running in the background that do not relate to MCMC sampling.

The MCMC results have been captured in a `Stanfit4Model` object named `jc_fit`. One may obtain summary statistics using the `print_stan_summary` function from the `bayes_stress_strain_utils` module, as follows. The elapsed time is printed as well:

```
bssu.print_stan_summary(jc_fit)
print('Elapsed time: {} s'.format(elapsed_time))
```

The output from the previous code is shown:

```
0.0 of 4000 iterations ended with a divergence (0.0%)
0 of 4000 iterations saturated the maximum tree depth of 10 (0.0%)
E-BFMI indicated no pathological behavior
Inference for Stan model: anon_model_5ba595b82fabba44dd5db1896f739604.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

               mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
A            780.62    0.01   0.71  779.2 780.14 780.62 781.13 781.99   4000    1.0
B            779.91    0.01   0.83 778.27 779.36  779.9 780.48 781.51   4000    1.0
n              0.11  4.4e-6 2.7e-4   0.11   0.11   0.11   0.11   0.11   3726    1.0
C            4.0e-3  6.7e-7 3.0e-5 3.9e-3 4.0e-3 4.0e-3 4.0e-3 4.0e-3   1942    1.0
m               1.0  4.3e-5 2.4e-3    1.0    1.0    1.0    1.0   1.01   3230    1.0
sd_sigma[0]    0.98  1.8e-3   0.08   0.84   0.93   0.98   1.03   1.16   1892    1.0
sd_sigma[1]    9.74    0.01   0.52    8.8   9.36    9.7  10.08  10.84   2022    1.0
lp__         -602.9    0.05   1.95 -607.7 -603.9 -602.5 -601.5 -600.1   1680    1.0

Samples were drawn using NUTS at Wed Aug  1 10:45:37 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Elapsed time: 21.356467493984383 s
```

The summary begins with the results of checks on the results from the MCMC run, all of which have passed.[33,34]

After these checks come a table of summary statistics. In this table, 0-based indexing is used, so `sd_sigma[0]` and `sd_sigma[1]` correspond to `sd_sigma[1]` and `sd_sigma[2]` in the Stan specification file. (However, in the future, PyStan will use 1-based indexing for the names of Stan variables.[39]) As one can see from the table, the mean values of the parameters from the model fit are nearly the same as the parameter values in `theta_model` that produced the simulated data. There are also other things worth noting. First, by default, PyStan generates four chains, each with 2000 samples total, the first 1000 of which are discarded as "warmup" samples. This warmup is present because the first several samples in a chain may be a poor representation of the posterior distribution. Second, there is an additional "parameter" `lp__`, which is not really a parameter, but rather the natural logarithm of the posterior probability density. Third, there are a couple diagnostics printed in the table. One is the effective sample size (`n_eff`), which indicates how ef-

fectively the posterior has been sampled. The values of this diagnostic range from about 1000 to 4000, which is reasonable. The other is the potential scale reduction factor (`Rhat`), which indicates if the distribution from which the samples are taken is close enough to the actual posterior distribution. Here, the diagnostics indicate that the MCMC sampling went well.

5.4 Testing Johnson-Cook Model with PyMC3

Before testing the Bayesian model for the Johnson-Cook strength model, one should check that the module that defines this model—namely, `jc_pymc3.py`, shown in Appendix C—can even be imported. However, this should *not* be done in a Jupyter notebook, but rather from a separate, fresh Python prompt that one has not yet used for anything else. If one gets the error message, "`To use MKL 2018 with Theano you MUST set MKL_THREADING_LAYER=GNU in your environment`," then one should exit that Python session, and in a different Python session (which *may* be a Jupyter notebook), one should run the following commands to set the environment variable `MKL_THREADING_LAYER`:

```
import os
os.environ["MKL_THREADING_LAYER"] = "GNU"
```

Note that if one attempts to reload the `jc_pymc3` module in the same Python session that generated errors, one is likely to see error messages such as "`AttributeError: module 'theano' has no attribute 'compile'`". However, if one has followed the previous instructions, the module defining the Bayesian model should be successfully importable.

The simulated data may be read in using the `read_from_pickle_file` from the module `bayes_stress_strain_utils` as follows. Since this module imports `pymc3` if it is available, the `MKL_THREADING_LAYER` must be set as described before:

```
import os
os.environ["MKL_THREADING_LAYER"] = "GNU"

import bayes_stress_strain_utils as bssu

my_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files", "jc_simulated_data_pymc3.pkl.gz"))
```

At this point, one may actually instantiate the model using the simulated data as

follows:

```
import jc_pymc3

jc_model = jc_pymc3.make_jc_model(my_data["epsilon_p"],
                                  my_data["sigma"],
                                  my_data["epsilon_p_dot"],
                                  my_data["T"],
                                  my_data["T_melt"],
                                  my_data["T_room"],
                                  my_data["epsilon_p_dot_0"],
                                  my_data["prior_params"])
```

One may now attempt to run MCMC sampling by using the function `sample` from the `pymc3` module on the model. In the following code, `try` and `except`, along with the `traceback` module, are used to capture Python exceptions and direct the resulting error messages from them to a file named `jc_trace_errs.txt`. (This helps prevent exceptions from stopping the execution of cells in a Jupyter notebook that occur after the exception.) The "`with jc_model:`" clause causes the `sample` function to run MCMC with the object `jc_model`. For the sake of reproducibility of the MCMC results, the seed for random number generation in MCMC sampling has been set via the `random_seed` argument of the `sample` function. The values for the arguments `draws` and `tune` in the `pm.sample` function are set so that the number of samples used for warmup (or "tuning" as it is called in PyMC3 documentation[3]) and final sampling in PyMC3 are the same as they are in Stan, and the values for the arguments `chains` and `cores` are chosen so that the number of chains generated in parallel is the same for both PyMC3 and Stan:

```
import pymc3 as pm
import traceback

try:
    with jc_model:
        jc_trace = pm.sample(draws = 1000, tune = 1000,
                             chains = 4, cores = 4,
                             random_seed = 12345)

except:
    err_filename = "jc_trace_errs.txt"
    print("Exception encountered: see file {}!".format(err_filename))
    with open(err_filename, "w") as err_file:
        traceback.print_exc(file=err_file)
```

The output from the PyMC3 run is shown:

```
INFO:pymc3:Auto-assigning NUTS sampler...
```

```
INFO:pymc3:Initializing NUTS using jitter+adapt_diag...
INFO:pymc3:Multiprocess sampling (4 chains in 4 jobs)
INFO:pymc3:NUTS: [sd_sigma, m, C, n, B, A]
Sampling 4 chains:  23%|██        | 1828/8000 [00:00<00:02, 2268.46draws/s]


Exception encountered: see file jc_trace_errs.txt!
```

In `jc_trace_errs.txt`, one may see the error message, "`Mass matrix contains zeros on the diagonal`". One way to solve this is to set initial values for the model parameters when rerunning MCMC, as shown in the following code. Here, the NumPy function `asarray` is used so that PyMC3 can do array arithmetic on `start_vals["sd_sigma_guess_mean"]`. Also, any error messages from exceptions are redirected to a new file, `jc_trace_errs_2.txt`:

```
import numpy as np

prior_params = my_data["prior_params"]

start_vals = {
    'A': prior_params['A_guess_mean'],
    'B': prior_params['B_guess_mean'],
    'n': prior_params['n_alpha']/
            (prior_params['n_alpha'] + prior_params['n_beta']),
    'C': prior_params['C_guess_mean'],
    'm': prior_params['m_guess_mean'],
    'sd_sigma': np.asarray(prior_params['sd_sigma_guess_mean'])
}

try:
    with jc_model:
        jc_trace = pm.sample(draws = 1000, tune = 1000,
                             chains = 4, cores = 4,
                             random_seed = 12345,
                             start = start_vals)

except:
    err_filename = "jc_trace_errs2.txt"
    print("Exception encountered: see file {}!".format(err_filename))
    with open(err_filename, "w") as err_file:
        traceback.print_exc(file=err_file)
```

The output of the new MCMC run is as follows:

```
INFO:pymc3:Auto-assigning NUTS sampler...
INFO:pymc3:Initializing NUTS using jitter+adapt_diag...
INFO:pymc3:Multiprocess sampling (4 chains in 4 jobs)
INFO:pymc3:NUTS: [sd_sigma, m, C, n, B, A]
Sampling 4 chains: 100%|██████████| 8000/8000 [00:33<00:00, 238.76draws/s]
```

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/mkl_fft/_numpy_fft.py:1044:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
array index, `arr[np.array(seq)]`, which will result either in an error or a different
result.
  output = mkl_fft.rfftn_numpy(a, s, axes)
```

Results from the MCMC run are captured in the PyMC trace object `jc_trace`. A table of summary statistics from this trace can be displayed as follows:

```
pm.summary(jc_trace)
```

The following are the summary statistics:

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/mkl_fft/_numpy_fft.py:1044:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
array index, `arr[np.array(seq)]`, which will result either in an error or a different
result.
  output = mkl_fft.rfftn_numpy(a, s, axes)
```

	mean	sd	mc_error	hpd_2.5	hpd_97.5	\
A	780.636184	0.724189	1.477554e-02	779.121581	782.011182	
B	779.915885	0.828828	1.755739e-02	778.276324	781.525927	
n	0.106390	0.000267	4.985844e-06	0.105862	0.106903	
C	0.003985	0.000030	4.943893e-07	0.003928	0.004046	
m	1.001189	0.002390	3.804957e-05	0.996607	1.005857	
sd_sigma__0	0.984831	0.076488	1.061820e-03	0.841532	1.136638	
sd_sigma__1	9.709786	0.499880	7.697319e-03	8.809314	10.763712	

	n_eff	Rhat
A	2486.828804	0.999695
B	2642.129683	0.999510
n	3235.550316	0.999814
C	3717.312155	0.999977
m	3902.032164	0.999889
sd_sigma__0	4266.818343	0.999773
sd_sigma__1	3987.109091	1.000199

As one can see, the mean values of the parameters from the model fit are nearly the same as the parameter values in `theta_model` that produced the simulated data. The quantities `n_eff` and `Rhat` are the same as in RStan and PyStan: the effective sample size and the potential scale reduction factor. PyMC3 would have warned if `n_eff` were too low. The diagnostic `Rhat` should be close to 1, and it is.

The performance here appears to be roughly that of a similar Stan model, with about 15 to 20 s for Stan and about 30 to 40 s for PyMC3.

5.5 Making Simulated Data for Zerilli-Armstrong (BCC) Model

As with the Johnson-Cook model, simulated stress-strain curves are to be created for several combinations of plastic strain rate and initial sample temperature, such as those in the following Python lists:

```
epsilon_p_dot = [2500.0, 0.001, 0.001,   0.1, 3500.0, 7000.0,
                 3000.0, 3000.0, 3500.0]
T_init =        [  77.0,  77.0, 298.0, 298.0,  298.0,  298.0,
                  473.0,  673.0,  873.0]
```

The values in these lists are taken from Gray et al.[9] They are in units of 1/s and Kelvin, respectively. As with the Johnson-Cook model, to account for the temperature rise during deformation of the sample, the sample density ρ and the specific heat as a function of temperature $c(T)$ are needed. Density ρ is again trivially represented by the Python variable `rho` as follows. The specific heat function is generated with the function `gen_lin_interp_func` from the Python module `bayes_stress_strain_utils` in Appendix C.

```
import bayes_stress_strain_utils as bssu
import os

rho = 7840.0 # kg/m^3

# Conversion factor from MPa to Pa
MPa_to_Pa = 1e6

c_func = bssu.gen_lin_interp_func(
    os.path.join(os.pardir, "Other_data",
                 "Austin_Specific_Heat_BCC_Iron.csv"),
    conv_func_y = lambda y: y/MPa_to_Pa,
    sep = ",",
    header = None
)
```

The function `gen_lin_interp_func` encapsulates most of the steps used to obtain `c_func` in Section 5.2. To account for the simulated data being in units of megapascals, the argument `conv_func_y` is used to divide the second column of data in `Austin_Specific_Heat_BCC_Iron.csv` by the conversion factor from megapascals to pascals, 10^6. The argument "`sep = ","`" accounts for the specific heat data file being in CSV format, and the "`header = None`" argument takes into account that the data file has no column headers. Both of these arguments correspond to the arguments "`sep = ","`" and "`header = None`" in the call

to the `pandas.read_table` function in Section 5.2.*

To create simulated data for the Zerilli-Armstrong (BCC) model, one needs a Python function specifying this model, and the function `za_bcc` from the module file `za_bcc.py` in Appendix C is used for this purpose. As has been done with the Johnson-Cook model, a wrapper function for `za_bcc` is used as the first argument to the function `simulate_data`. This wrapper function is as follows:

```
from za_bcc import za_bcc
import numpy as np

def sigma_model_func(epsilon_p,
                     epsilon_p_dot,
                     T,
                     theta_model):

    return za_bcc(
        epsilon_p,
        np.log(epsilon_p_dot),
        T,
        theta_model["C0"],
        theta_model["C1"],
        theta_model["C3"],
        theta_model["C4"],
        theta_model["C5"],
        theta_model["n"]
    )
```

The model parameters needed by `sigma_model_func` are the following:

```
theta_model = {
    "C0": 50.0,   # MPa
    "C1": 1800.0, # MPa
    "C3": 0.0015,
    "C4": 0.000045,
    "C5": 1200.0, # MPa
    "n": 0.62
}
```

The values of these parameters happen to be from Gray et al.,[9] but in principle, they could be set to any plausible values. At this point, nearly all the information needed for `simulate_data` has been input to a Python session. The maximum value for ϵ_p and values for $SD_{\sigma,1}$ and $SD_{\sigma,2}$ are set as follows:

```
epsilon_p_max = 0.2
sd_sigma = [1.0, 10.0]
```

*The function `gen_lin_interp_func` actually passes "`sep = ","`" and "`header = None`" to the `pandas.read_table` function.

Finally, the simulated data can be generated as follows. Again, `beta_TQ`, the Taylor-Quinney coefficient, is set to zero for low strain rates to simulate the lack of a temperature rise at those rates. Similarly, `curr_sd_sigma` is either $SD_{\sigma,1}$ or $SD_{\sigma,2}$, depending on the strain rate. Again, for the sake of reproducibility, the following code sets the seed used by functions in the NumPy submodule `random`.

```
import bayes_stress_strain_utils as bssu

## Initializing to empty lists
sigma = []
epsilon_p = []
T = []

min_curve_size = 40
max_curve_size = 50

np.random.seed(12345)

for i in range(len(epsilon_p_dot)):

    if epsilon_p_dot[i] <= 1.0:
        beta_TQ = 0.0
        curr_sd_sigma = sd_sigma[0]
    else:
        beta_TQ = 0.9
        curr_sd_sigma = sd_sigma[1]

    # Sets curve_size to a random integer between min_curve_size
    # and max_curve_size
    curve_size = np.random.randint(min_curve_size,
                                   max_curve_size + 1)

    curr_data = bssu.simulate_data(
        sigma_model_func,
        epsilon_p_max,
        epsilon_p_dot[i],
        T_init[i],
        theta_model,
        beta_TQ,
        rho,
        c_func,
        curve_size
    )

    sigma.append(curr_data["sigma"] +
                 curr_sd_sigma*np.random.randn(curve_size))

    epsilon_p.append(curr_data["epsilon_p"])
    T.append(curr_data["T"])
```

Again, the simulated data is plotted to a ".pdf" file as a sanity check (via the function

`plot_stress_strain_curves`). A plot of this simulated data is shown in Fig. 10.

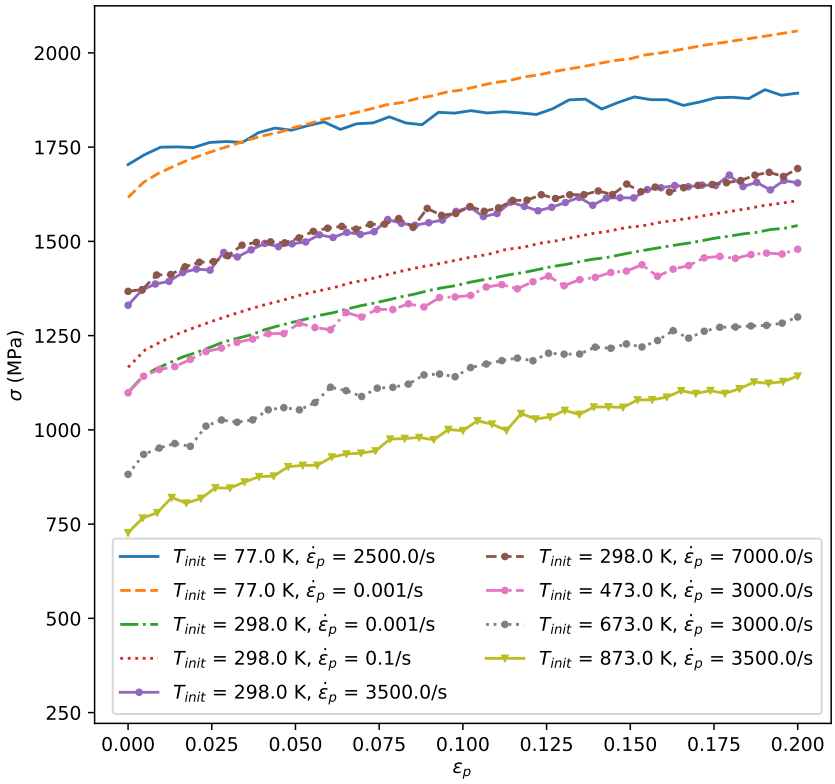


Fig. 10 Plot of simulated data used to test the Zerilli-Armstrong (BCC) PyStan and PyMC3 models

At this point, the simulated data can begin to be converted into forms more suitable for PyStan and PyMC3 and then saved to Gzip-compresssed[29] Python pickle files. Both of these need variables pertaining to the priors, that is, `C0_guess_mean`, `n_alpha`, and so on, so these are set using the JSON file `ZA_BCC_priors.json` from Section 4:

```
prior_params = bssu.read_from_json_file(
    os.path.join(os.pardir, "Other_data", "ZA_BCC_priors.json")
)
```

As with the data for the Johnson-Cook model for PyStan, input for MCMC sam-

pling needs to be a dictionary whose keys are the variable names in the `data` block of a Stan specification file, and this dictionary is saved to a Python pickle file. Again, `np.concatenate` is used to put the strain, stress, and temperature data in the form of the vectors specified by `za_bcc.stan` in Appendix D:

```
dict_for_pystan = {
    "num_curves": len(epsilon_p_dot),
    "curve_sizes": [len(s) for s in sigma],
    "epsilon_p_dot": epsilon_p_dot,
    "epsilon_p": np.concatenate(epsilon_p),
    "sigma": np.concatenate(sigma),
    "T": np.concatenate(T)
}

dict_for_pystan.update(prior_params)

bssu.save_to_pickle_file(
    dict_for_pystan,
    os.path.join("pkl_data_files",
                 "za_bcc_simulated_data_pystan.pkl.gz")
)
```

For PyMC3, the data to be saved are what are to be passed as arguments to `make_za_bcc_model` in the module `za_bcc_pymc3.py` shown in Appendix C. As with the data for the Johnson-Cook model for PyMC3, these arguments are placed in a dictionary and saved as follows:

```
bssu.save_to_pickle_file({
        "epsilon_p_dot": epsilon_p_dot,
        "epsilon_p": epsilon_p,
        "sigma": sigma,
        "T": T,
        "prior_params": prior_params
    },
    os.path.join("pkl_data_files",
                 "za_bcc_simulated_data_pymc3.pkl.gz")
)
```

5.6 Testing Zerilli-Armstrong (BCC) Model with PyStan

The process for compiling the Zerilli-Armstrong (BCC) model is nearly the same as the corresponding one for the Johnson-Cook model. One imports the PyStan module, compiles the appropriate Stan specification file (i.e., `za_bcc.stan`) into a `StanModel` object, and then saves the resulting object to a Python pickle file. As with the Johnson-Cook model, saving to the pickle is done with the convenience function `save_to_pickle_file` from the module

bayes_stress_strain_utils.py:

```
import pystan
import bayes_stress_strain_utils as bssu
import os

za_bcc_model = pystan.StanModel(
    os.path.join(os.pardir, "stan_model_specs",
                 "za_bcc.stan"))

bssu.save_to_pickle_file(za_bcc_model,
                         os.path.join("compiled_stan_models",
                                      "za_bcc.pkl"))
```

Again, the Python pickle file for a `StanModel` object (here `za_bcc.pkl`) only works with Python sessions done on the same system used to generate the pickle file, or at least a system that is nearly identical.

To run MCMC, the simulated data may be loaded into the Python session, and the Zerilli-Armstrong (BCC) model may be loaded as well, if it has not been loaded already:

```
import bayes_stress_strain_utils as bssu
import os.path

my_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files",
                 "za_bcc_simulated_data_pystan.pkl.gz"))

za_bcc_model = bssu.read_from_pickle_file(
    os.path.join("compiled_stan_models", "za_bcc.pkl"))
```

Again, one fits the model to the simulated data via the `sampling` method and prints summary statistics from the fit. For the sake of reproducibility, the seed for random number generation is again set via the `seed` argument of the `sampling` method. As with the MCMC runs for the Johnson-Cook model, since the only way PyStan prints elapsed time is via terminal output (which may not be visible in, for example, a Jupyter notebook), the functionality of Python's `time` module is used to estimate the elapsed time in seconds:

```
import time

start_time = time.perf_counter()
za_bcc_fit = za_bcc_model.sampling(data = my_data,
                                   seed = 12345)
elapsed_time = time.perf_counter() - start_time
```

```
bssu.print_stan_summary(za_bcc_fit)
print('Elapsed time: {} s'.format(elapsed_time))
```

The following are the warnings and summary statistics from the MCMC run:

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/pystan/misc.py:399:
FutureWarning: Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  elif np.issubdtype(np.asarray(v).dtype, float):


482.0 of 4000 iterations ended with a divergence (12.05%)
  Try running with larger adapt_delta to remove the divergences
8 of 4000 iterations saturated the maximum tree depth of 10 (0.2%)
  Run again with max_depth set to a larger value to avoid saturation
Chain 0: E-BFMI = 0.007345947086255753
  E-BFMI below 0.2 indicates you may need to reparameterize your model
Inference for Stan model: anon_model_94ebba402ea6796923b7e03a53b54c34.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

               mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
C0            33.57   14.56  20.59   0.23   7.71  41.37  48.58  59.56      2    3.0
C1           1353.9  552.71 781.65   0.31 801.51 1801.5 1808.3 1820.9      2 120.46
C3             1.61    1.97   2.78 1.5e-3 1.5e-3 1.5e-3   3.53   6.43      2  3.5e4
C4             0.19    0.23   0.33 4.4e-5 4.5e-5 4.5e-5   0.41   0.75      2  2.9e4
C5           900.95  367.54 519.78   0.79 537.62 1200.2 1201.8 1204.4      2 340.76
n              0.58    0.04   0.06   0.47   0.54   0.62   0.62   0.62      2  63.86
sd_sigma[0]    0.81    0.17   0.23   0.42   0.58   0.92   0.97   1.06      2   4.94
sd_sigma[1]    7.68    2.59   3.67   1.36   4.54    9.6   9.95   10.6      2  10.58
lp__       -1.4e70  1.8e70 3.1e70-8.8e70-1.7e70 -888.6 -886.9 -885.1      3    2.8

Samples were drawn using NUTS at Mon Aug  6 14:30:01 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Elapsed time: 104.2918440940557 s
```

Obviously, these results are poor. The mean values of most of the parameters are nowhere near what they should be, and the potential scale reduction factor indicates a lack of convergence to the correct posterior. The low effective sample size indicates that the posterior has hardly been sampled, which suggests a bad starting point for the sampling.

The fix for this is to initialize the sampling from some reasonable initial values. The mean values of the priors are good enough for this:

```
init_values = {
    "C0": my_data["C0_guess_mean"],
    "C1": my_data["C1_guess_mean"],
    "C3": my_data["C3_guess_mean"],
    "C4": my_data["C4_guess_mean"],
    "C5": my_data["C5_guess_mean"],
    "n": my_data["n_alpha"]/(my_data["n_alpha"] + my_data["n_beta"])
}
```

However, if one looks at the reference documentation for PyStan,[32] it says that the initial values should be either a list of Python dictionaries, where each element in the list is a dictionary of initial values for a chain, or a function that returns a dictionary of initial values. Accordingly, the list of initial values passed to the `sampling` method is as follows:

```
num_chains = 4
init_values_list = [init_values]*num_chains
```

The MCMC sampling can then proceed as shown. Here, the number of chains is explicitly set to the length of `init_values_list` for the sake of consistency:

```
start_time = time.perf_counter()
za_bcc_fit = za_bcc_model.sampling(data = my_data,
                                   seed = 12345,
                                   init = init_values_list,
                                   chains = len(init_values_list))
elapsed_time = time.perf_counter() - start_time

bssu.print_stan_summary(za_bcc_fit)
print('Elapsed time: {} s'.format(elapsed_time))
```

The following are the output and summary statistics from the new MCMC run:

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/pystan/misc.py:399:
FutureWarning: Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  elif np.issubdtype(np.asarray(v).dtype, float):


0.0 of 4000 iterations ended with a divergence (0.0%)
113 of 4000 iterations saturated the maximum tree depth of 10 (2.825%)
  Run again with max_depth set to a larger value to avoid saturation
E-BFMI indicated no pathological behavior
Inference for Stan model: anon_model_94ebba402ea6796923b7e03a53b54c34.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

              mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
```

```
C0           44.93    0.31   8.15  28.43  39.65  45.12  50.31  60.66    681    1.01
C1          1804.9     0.3   7.76 1790.0 1799.8 1804.7 1809.9 1820.5    683    1.01
C3          1.5e-3  3.7e-7 9.5e-6 1.5e-3 1.5e-3 1.5e-3 1.5e-3 1.5e-3    682    1.01
C4          4.5e-5  9.1e-9 2.8e-7 4.4e-5 4.5e-5 4.5e-5 4.5e-5 4.5e-5    985    1.01
C5          1201.0    0.04   1.91 1197.2 1199.7 1201.0 1202.3 1204.6   2698     1.0
n             0.62  2.5e-5 1.3e-3   0.62   0.62   0.62   0.62   0.62   2510     1.0
sd_sigma[0]   0.94  1.2e-3   0.06   0.84    0.9   0.94   0.98   1.07   2436     1.0
sd_sigma[1]   9.77  8.2e-3   0.43   8.99   9.47   9.74  10.04  10.64   2677     1.0
lp__        -888.0    0.06   2.04 -892.8 -889.1 -887.7 -886.5 -885.0   1211     1.0

Samples were drawn using NUTS at Mon Aug  6 14:31:42 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Elapsed time: 100.77757640404161 s
```

As one can see, the mean values of the parameters from the model fit are nearly the same as the parameter values from `theta_model` that produced the simulated data, and the effective sample sizes and potential scale reduction factors are reasonable. However, there is a warning about the maximum treedepth and advice on how fix the issue by increasing the parameter `max_treedepth`. This advice is followed as shown:

```
start_time = time.perf_counter()
za_bcc_fit = za_bcc_model.sampling(data = my_data,
                                   seed = 12345,
                                   init = init_values_list,
                                   control = {'max_treedepth': 15},
                                   chains = len(init_values_list))
elapsed_time = time.perf_counter() - start_time

bssu.print_stan_summary(za_bcc_fit)
print('Elapsed time: {} s'.format(elapsed_time))
```

The following are the summary statistics from the MCMC run:

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/pystan/misc.py:399:
FutureWarning: Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  elif np.issubdtype(np.asarray(v).dtype, float):


0.0 of 4000 iterations ended with a divergence (0.0%)
0 of 4000 iterations saturated the maximum tree depth of 15 (0.0%)
E-BFMI indicated no pathological behavior
Inference for Stan model: anon_model_94ebba402ea6796923b7e03a53b54c34.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.
```

```
               mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
C0             44.8    0.26   8.37   28.3  39.17  44.73  50.42  61.49   1008    1.0
C1           1805.0    0.25   7.96 1789.2 1799.6 1805.0 1810.4 1820.7   1007    1.0
C3           1.5e-3  3.1e-7 9.8e-6 1.5e-3 1.5e-3 1.5e-3 1.5e-3 1.5e-3   1013    1.0
C4           4.5e-5  8.7e-9 2.9e-7 4.4e-5 4.5e-5 4.5e-5 4.5e-5 4.5e-5   1107    1.0
C5           1201.0    0.04   1.89 1197.3 1199.7 1201.0 1202.2 1204.7   2378    1.0
n              0.62  2.4e-5 1.2e-3   0.62   0.62   0.62   0.62   0.62   2684    1.0
sd_sigma[0]    0.94  1.3e-3   0.06   0.83    0.9   0.94   0.98   1.07   2306    1.0
sd_sigma[1]     9.8    0.01   0.42   8.99    9.5   9.79  10.07  10.65   1767    1.0
lp__         -888.0    0.05    2.0 -892.7 -889.1 -887.7 -886.5 -885.0   1364    1.0

Samples were drawn using NUTS at Mon Aug  6 14:33:33 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Elapsed time: 110.85108313290402 s
```

At this point, both the mean parameter values and the diagnostics are returning reasonable results.

5.7 Testing Zerilli-Armstrong (BCC) Model with PyMC3

The simulated data may be read in using the `read_from_pickle_file` function from the module `bayes_stress_strain_utils`, and the Zerilli-Armstrong (BCC) model can then be instantiated from the data:

```
import os

# This must be set before loading za_bcc_pymc3!
os.environ["MKL_THREADING_LAYER"] = "GNU"

import za_bcc_pymc3
import bayes_stress_strain_utils as bssu

my_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files",
                 "za_bcc_simulated_data_pymc3.pkl.gz"))

za_bcc_model = za_bcc_pymc3.make_za_bcc_model(my_data["epsilon_p"],
                                              my_data["sigma"],
                                              my_data["epsilon_p_dot"],
                                              my_data["T"],
                                              my_data["prior_params"])
```

One may now attempt to run MCMC sampling by using the PyMC3 function `sample` with the model. Here, `try` and `except` are used as they are with the Johnson-Cook PyMC3 model, as are the `sample` function arguments `draws`,

tune, chains, cores, and random_seed:

```
import pymc3 as pm
import traceback

try:
    with za_bcc_model:
        za_bcc_trace = pm.sample(draws = 1000, tune = 1000,
                                 chains = 4, cores = 4,
                                 random_seed = 12345)

except:
    err_filename = "za_bcc_trace_errs.txt"
    print("Exception encountered: see file {}!".format(err_filename))
    with open(err_filename, "w") as err_file:
        traceback.print_exc(file=err_file)
```

The following is the output from the MCMC run:

```
Auto-assigning NUTS sampler...
INFO:pymc3:Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
INFO:pymc3:Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
INFO:pymc3:Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
INFO:pymc3:NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
Sampling 4 chains:   0%|          | 0/8000 [00:00<?, ?draws/s]


Exception encountered: see file za_bcc_trace_errs.txt!
```

In za_bcc_trace_errs.txt, one would likely see the error message, "ValueError: Bad initial energy: inf. The model might be misspecified." One way to solve this is to set initial values for the model parameters when rerunning MCMC, as shown in the following code. In this code, np.asarray is used so that PyMC3 can do array arithmetic on start_vals["sd_sigma_guess_mean"]. Also, any error messages are redirected to za_bcc_trace_errs_2.txt:

```
import numpy as np

prior_params = my_data["prior_params"]

start_vals = {
    'C0': prior_params['C0_guess_mean'],
    'C1': prior_params['C1_guess_mean'],
    'C3': prior_params['C3_guess_mean'],
```

```
    'C4': prior_params['C4_guess_mean'],
    'C5': prior_params['C5_guess_mean'],
    'n': prior_params['n_alpha']/
            (prior_params['n_alpha'] + prior_params['n_beta']),
    'sd_sigma': np.asarray(prior_params['sd_sigma_guess_mean'])
}

try:
    with za_bcc_model:
        za_bcc_trace = pm.sample(draws = 1000, tune = 1000,
                                 chains = 4, cores = 4,
                                 random_seed = 12345,
                                 start = start_vals)

except:
    err_filename = "za_bcc_trace_errs2.txt"
    print("Exception encountered: see file {}!".format(err_filename))
    with open(err_filename, "w") as err_file:
        traceback.print_exc(file=err_file)
```

The following is the output of the new MCMC run:

```
Auto-assigning NUTS sampler...
INFO:pymc3:Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
INFO:pymc3:Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
INFO:pymc3:Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
INFO:pymc3:NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
Sampling 4 chains: 100%|[redacted]| 8000/8000 [04:41<00:00, 10.28draws/s]
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/mkl_fft/_numpy_fft.py:1044:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
array index, `arr[np.array(seq)]`, which will result either in an error or a different
result.
  output = mkl_fft.rfftn_numpy(a, s, axes)
The acceptance probability does not match the target. It is 0.8838034197424812, but should
be close to 0.8. Try to increase the number of tuning steps.
WARNING:pymc3:The acceptance probability does not match the target. It is
0.8838034197424812, but should be close to 0.8. Try to increase the number of tuning
steps.
There were 55 divergences after tuning. Increase `target_accept` or reparameterize.
ERROR:pymc3:There were 55 divergences after tuning. Increase `target_accept` or
reparameterize.
The acceptance probability does not match the target. It is 0.8787013918835163, but should
be close to 0.8. Try to increase the number of tuning steps.
WARNING:pymc3:The acceptance probability does not match the target. It is
0.8787013918835163, but should be close to 0.8. Try to increase the number of tuning
steps.
There were 2 divergences after tuning. Increase `target_accept` or reparameterize.
ERROR:pymc3:There were 2 divergences after tuning. Increase `target_accept` or
reparameterize.
```

```
The number of effective samples is smaller than 25% for some parameters.
INFO:pymc3:The number of effective samples is smaller than 25% for some parameters.
```

There are warnings about divergences after tuneup, and per the advice in the warning, the NUTS parameter `target_accept` is increased to fix this issue:

```
try:
    with za_bcc_model:
        za_bcc_trace = pm.sample(draws = 1000, tune = 1000,
                                 chains = 4, cores = 4,
                                 random_seed = 12345,
                                 nuts_kwargs = {"target_accept": 0.9},
                                 start = start_vals)

except:
    err_filename = "za_bcc_trace_errs3.txt"
    print("Exception encountered: see file {}!".format(err_filename))
    with open(err_filename, "w") as err_file:
        traceback.print_exc(file=err_file)
```

The following is the output of this MCMC run:

```
Auto-assigning NUTS sampler...
INFO:pymc3:Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
INFO:pymc3:Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
INFO:pymc3:Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
INFO:pymc3:NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
Sampling 4 chains: 100%|[redacted]| 8000/8000 [06:18<00:00,  4.68draws/s]
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/mkl_fft/_numpy_fft.py:1044:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
array index, `arr[np.array(seq)]`, which will result either in an error or a different
result.
  output = mkl_fft.rfftn_numpy(a, s, axes)
```

At this point, there are no further warnings from PyMC3 itself, so summary statistics for this latest MCMC run are printed:

```
pm.summary(za_bcc_trace)
```

The following are the printed summary statistics:

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/mkl_fft/_numpy_fft.py:1044:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
```

```
array index, `arr[np.array(seq)]`, which will result either in an error or a different
result.
  output = mkl_fft.rfftn_numpy(a, s, axes)
```

	mean	sd	mc_error	hpd_2.5	\
C0	45.331548	8.336647e+00	2.349636e-01	28.798898	
C1	1804.546350	7.939917e+00	2.245410e-01	1788.838173	
C3	0.001495	9.745669e-06	2.742305e-07	0.001478	
C4	0.000045	2.883439e-07	8.366848e-09	0.000044	
C5	1201.006813	1.902154e+00	3.950585e-02	1197.282618	
n	0.620302	1.248886e-03	2.625048e-05	0.617890	
sd_sigma__0	0.945305	5.848015e-02	1.044262e-03	0.831415	
sd_sigma__1	9.783543	4.347707e-01	7.976654e-03	8.979167	

	hpd_97.5	n_eff	Rhat
C0	61.262020	1229.543702	0.999541
C1	1819.647527	1229.569541	0.999534
C3	0.001516	1248.887003	0.999543
C4	0.000045	1244.031659	0.999510
C5	1204.621064	2319.883038	1.000575
n	0.622796	2349.247640	1.000599
sd_sigma__0	1.055272	2939.571202	0.999745
sd_sigma__1	10.675993	2916.736998	0.999548

The diagnostics `n_eff` and `Rhat` diagnostics appear reasonable, and the mean values of the parameters are also close to the parameter values in `theta_model` used to generate the simulated data. As far as performance is concerned, an MCMC run of the Zerilli-Armstrong (BCC) model with PyMC3 appears to be about three to four times as long as a corresponding one with Stan, given the same simulated data.

6. Fitting Strength Models to Experimental Data

6.1 Functions for Fitting Models

Some of the custom functions in Section 5.1 are also used in the process of fitting models, in particular `gen_lin_interp_func`, `print_stan_summary`, `read_from_pickle_file`, and `save_to_pickle_file`. In addition to these, there are the following custom functions:

- `save_stan_fit_to_csv`, a function that saves the summary statistics and MCMC samples from a `StanFit4Model` object to CSV files;
- `save_pymc3_trace_to_csv`, a function that saves the summary statistics and MCMC samples from a PyMC3 trace to CSV files; and

- `calc_temps`, a function for estimating the temperatures at the points of a stress-strain curve.

The first two functions involve details pertaining to the functionality of PyStan and PyMC3, while the last one pertains more directly to the sample problem. Accordingly, these functions are examined in more detail. The arguments to the function `save_stan_fit_to_csv` are as follows:

- `fit`, a `StanFit4Model` object
- `summary_csv_filename`, the name of CSV file to which summary statistics are written
- `samples_csv_filename`, the name of CSV file to which MCMC samples are written. If the file ends in ".gz", it is Gzip-compressed.[29]

The first two statements in the body of this function simply ensure that any directories in the paths `summary_csv_filename` and `samples_csv_filename` actually exist, and create them if they do not already exist:

```
_ensure_path_to_file_exists(summary_csv_filename)
_ensure_path_to_file_exists(samples_csv_filename)
```

The function used in these statements, `_ensure_path_to_file_exists`, is an internal function defined in the module `bayes_stress_strain_utils`. The details of it may be mainly of interest to readers who are looking for example code to use as a reference.

The following Python statements write the summary statistics to a CSV file.

```
summary = fit.summary()

summary_df = pandas.DataFrame(summary["summary"],
                              index = summary["summary_rownames"],
                              columns = summary["summary_colnames"])

summary_df.to_csv(summary_csv_filename)
```

The first of these statements creates a dictionary named `summary` that contains the summary statistics. The second Python statement takes these dictionary values and creates from them a Pandas data frame named `summary_df`. The dictionary

value `summary["summary"]` is a 2-D NumPy array that contains the statistics themselves. This array becomes the numerical contents of the data frame. The value `summary["summary_rownames"]` contains labels for the rows of this array, which correspond to the parameter names shown in the summary statistics output from `print(fit)`, such as `A`, `B`, and so on, for the Johnson-Cook model, `C0`, `C1`, and so on, for the Zerilli-Armstrong (BCC) model, as well as nuisance parameters `sd_sigma[1]` and `sd_sigma[2]` and the pseudoparameter `lp__`. These row labels become the row labels of the data frame `summary_df`. The value `summary["summary_colnames"]` contains labels for the columns of the array of statistics, such as `mean`, `se_mean`, `Rhat`, and so on, which also appear in the output from `print(fit)`, and these labels become the column headers of `summary_df`. Finally, the last Python statement prints the contents of the data frame `summary_df` to a CSV file named `summary_csv_filename`.

The next part of the function indicates whether to compress the file used to save the MCMC samples, since that file can potentially be quite large:

```
if samples_csv_filename.endswith(".gz"):
    my_open = gzip.open
else:
    my_open = open
```

Here, the variable `my_open` represents the function used to open a file (and create it if it does not exist). If the filename `samples_csv_filename` ends in ".gz", then the variable is assigned to the function that opens Gzip-compressed files. Otherwise, it is assigned to the function that opens regular files.

Finally, the MCMC samples are written to a file:

```
samples = fit.extract(permuted = False)

with my_open(samples_csv_filename, "wb") as f:

    csv_header = ",".join(summary["summary_rownames"])
    f.write("{}\n".format(csv_header).encode("utf-8"))

    for chain_id in range(samples.shape[1]):
        np.savetxt(f, samples[:, chain_id, :], delimiter = ",")
```

The first of these Python statements creates a 3-D NumPy array named `samples`, due to the argument "`permuted = False`" of the `extract` method. The first dimension of this array is the number of MCMC samples. The second di-

mension is the number of chains, and the third is the number of parameters,[32] including the pseudoparameter `lp__`. Essentially, when `chain_id` is the integer identifier of a chain, then `samples[:, chain_id, :]` is a 2-D array where column `i` is a set of MCMC samples pertaining to the parameter named `summary["summary_rownames"][i]`. Accordingly, the column headers of the CSV file containing the samples (i.e., `csv_header`) are the strings in the array `summary["summary_rownames"]`. For each possible value of `chain_id`, the `for` loop below the `with` clause prints the contents of `samples[:, chain_id, :]` to a CSV file via the `savetxt` function of NumPy. Due to an implementation detail of this function, the CSV file named `samples_csv_filename` has to be opened in *binary* mode,[40] hence the argument `"wb"` of `my_open` and the use of the method `encode` to write the column headers to the CSV file. In future versions of PyStan, writing samples to a CSV file should become much simpler, since `StanFit4Model` objects will have a method that writes MCMC samples to Pandas data frames,[41] which can be readily written to CSV files.[36]

The arguments to `save_pymc3_trace_to_csv` are much like those of `save_stan_fit_to_csv`:

- `trace`, a PyMC3 trace object, which contains the results from a PyMC3 MCMC run
- `summary_csv_filename`, the name of CSV file to which summary statistics are written
- `samples_csv_filename`, the name of CSV file to which MCMC samples are written. If the file ends in ".gz", it is Gzip-compressed.[29]

As with `save_stan_fit_to_csv`, the first two statements in the body of this function simply ensure that any directories in the paths `summary_csv_filename` and `samples_csv_filename` actually exist, and create them if they do not already exist:

```
_ensure_path_to_file_exists(summary_csv_filename)
_ensure_path_to_file_exists(samples_csv_filename)
```

The function `_ensure_path_to_file_exists` is the same internal function mentioned previously.

The next Python statement writes the summary statistics to the file named `summary_csv_filename`:

```
pm.summary(trace).to_csv(summary_csv_filename)
```

Here, `pm.summary(trace)` is a Pandas data frame containing summary statistics, which are then written to a CSV file via the `to_csv` method. The `summary` function is from the PyMC3 module.

The next part of the function indicates whether to compress the file used to save the MCMC samples, since that file can potentially be quite large.

```
if samples_csv_filename.endswith(".gz"):
    compression = "gzip"
else:
    compression = None
```

Finally, the samples are written to a file as follows:

```
df = pm.trace_to_dataframe(trace)

df.to_csv(samples_csv_filename,
          compression = compression, index = False)
```

The PyMC3 function `trace_to_dataframe` is used to extract the MCMC samples to the data frame `df`, where each column of the data frame is the sequence of MCMC samples for a particular model parameter. In the method `to_csv`, the keyword argument `compression = compression`, where the variable `compression` (on the right-hand side of the equals sign) has been defined to be either `"gzip"` or `None`, indicates the type of compression to be applied to the contents of the CSV file. The argument "`index = False`" indicates that there is not to be an unnecessary additional column that numbers the rows in the CSV file.

The arguments to `calc_temps` are as follows:

- `T_init`, the initial temperature of the sample
- `epsilon_p`, the sequence of plastic strains in the stress-strain curve
- `sigma`, the sequence of stresses in the stress-strain curve

- `f_area`, the parameter f_{area} from Eq. 2
- `beta_TQ`, the Taylor-Quinney coefficient
- `rho`, the density of the sample
- `specific_heat_func`, a function that returns the specific heat for a given temperature (which can and later on is generated using `gen_lin_interp_func`)

The first few lines of the body of this function are these:

```
curve_size = len(epsilon_p)
T = np.empty(curve_size)

T[0] = T_init + beta_TQ*f_area*sigma[0]*epsilon_p[0]/(
    rho*specific_heat_func(T_init))
```

The first statement simply sets a descriptively named variable, `curve_size`, to the length of the list or 1-D array `epsilon_p`, which is the length of the stress-strain curve under consideration. The next statement initializes the 1-D array of temperatures so that it has the correct length. The last statement corresponds to Eq. 2, but with $T_{init}^{i_c}$ (i.e., `T_init`) moved to the right-hand side. (No variable corresponding to index i_c appears in `calc_temps`, since the value of i_c is effectively fixed by the choice of `epsilon_p` and `sigma`.)

The rest of the function body is as follows:

```
for i in range(1, curve_size):

    # Using trapezoid rule to estimate area under stress-strain
    # curve over interval [epsilon_p[i-1], epsilon_p[i]].
    area_under_curve = 0.5*(sigma[i-1] + sigma[i])*(
        epsilon_p[i] - epsilon_p[i-1])

    T_rise = beta_TQ*area_under_curve/(
        rho*specific_heat_func(T[i-1]))

    T[i] = T[i-1] + T_rise

return T
```

The body of the `for` loop in the function body corresponds to Eq. 1. The first statement estimates the integral in that equation (i.e., `area_under_curve`) via the trapezoid rule of numerical integration.[42] Once this integral is calculated, the

temperature rise $T_j^{i_c} - T_{j-1}^{i_c}$ (or `T_rise`) may be determined. The temperature of the current data point $T_j^{i_c}$ (or `T[i]`) is then the sum of the temperature rise and the temperature of the previous data point $T_{j-1}^{i_c}$ (or `T[i-1]`).

The very last line of the function body, of course, returns the array of temperatures from the function.

6.2 Preprocessing Experimental Data

Several of the data files from Section 4 are to be read into Python and then processed into pickle files that are used in later analyses. First, the MIDAS data files are read in as follows:

```
import pandas
import os
import numpy as np

T_init_str        = [   "77",   "77",   "298", "298",  "298",  "298",
                      "473",  "673",  "873"]
epsilon_p_dot_str = ["0.001", "2500", "0.001", "0.1", "3500", "7000",
                     "3000", "3000", "3500"]

epsilon_p = []
sigma = []

for i in range(len(T_init_str)):
    csv_filename = "T{}K_edot{}_per_s.csv".format(
        T_init_str[i],
        epsilon_p_dot_str[i])

    out_data = pandas.read_csv(
        os.path.join(os.pardir, "MIDAS_data", csv_filename),
        header = None
    )

    # The first column of out_data is the strain.
    epsilon_p.append(np.asarray(out_data.iloc[:,0]))

    # The second column of out_data is the stress.
    sigma.append(np.asarray(out_data.iloc[:,1]))
```

This Python code is somewhat similar to the code used to generate simulated data, in that initial temperatures and strain rates are specified, and lists of vectors containing strains and stresses are built up. Of course, in place of the function `simulate_data` is the function `pandas.read_csv`, which reads experimental data from a CSV file. The argument "`header = None`" in the call to `pandas.read_csv` prevents the function from mistaking the first line of the

CSV file for column headers.

To calculate temperatures, the density ρ, specific heat $c(T)$, Taylor-Quinney coefficient β_{TQ}, and f_{area} are needed. The first two of these are determined from known data and can be specified as follows:

```
import bayes_stress_strain_utils as bssu

rho = 7840.0 # kg/m^3

# Conversion factor from MPa to Pa
MPa_to_Pa = 1e6

c_func = bssu.gen_lin_interp_func(
    os.path.join(os.pardir, "Other_data",
                 "Austin_Specific_Heat_BCC_Iron.csv"),
    conv_func_y = lambda y: y/MPa_to_Pa,
    sep = ",",
    header = None
)
```

As pointed out in Section 1, the next two quantities are more uncertain, so temperature calculations are done for a few combinations of reasonable estimates of β_{TQ} and f_{area} (shown in Table 1):

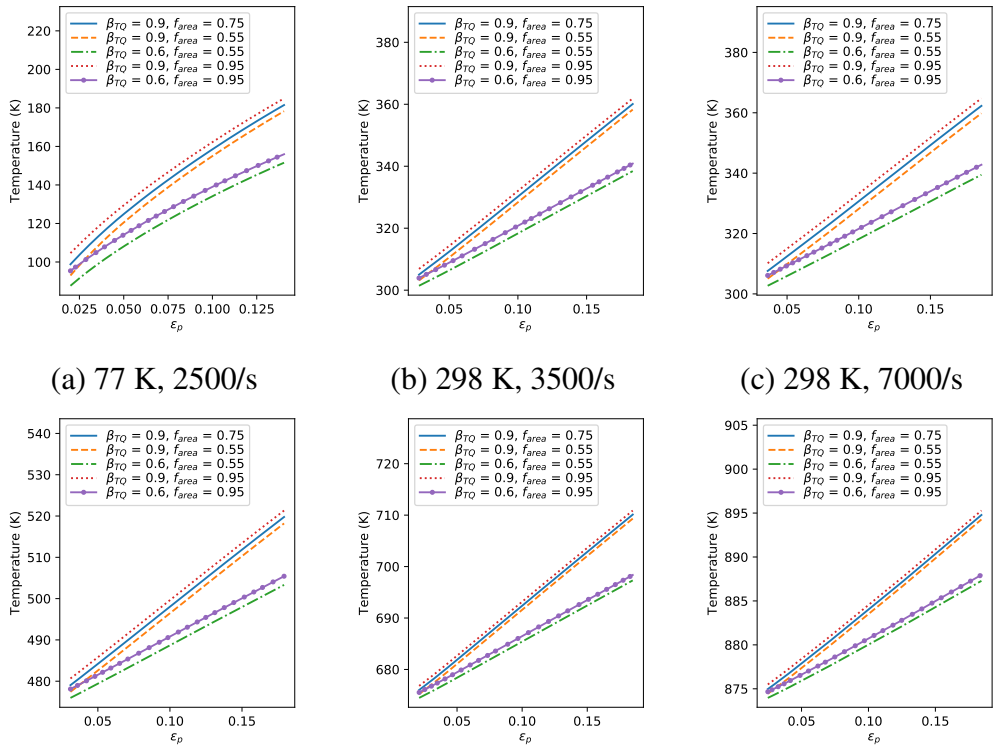
```
T = {}

beta_TQ_f_area = [(0.9, 0.75),
                  (0.9, 0.55),
                  (0.6, 0.55),
                  (0.9, 0.95),
                  (0.6, 0.95)]

# The function "float" is used here to convert strings to their
# corresponding numerical values, i.e. "0.1" to 0.1.
epsilon_p_dot = [float(epdot) for epdot in epsilon_p_dot_str]
T_init = [float(Ti) for Ti in T_init_str]

for bTQ_fA in beta_TQ_f_area:
    T[bTQ_fA] = []

    for i in range(len(epsilon_p_dot)):
        if epsilon_p_dot[i] > 1.0:
            bTQ, fA = bTQ_fA
            T[bTQ_fA].append(
                bssu.calc_temps(T_init[i], epsilon_p[i], sigma[i],
                                fA, bTQ, rho, c_func))
        else:
            curve_size = len(sigma[i])
            T[bTQ_fA].append(np.full(curve_size, T_init[i]))
```

The variable `T` here is a dictionary that uses the tuples `(0.9, 0.75)`, `(0.9, 0.55)`, and so on, as keys, where the first element of the tuple is a value of β_{TQ}, and the second is a value of f_{area}. The value associated with each key is a list of arrays of temperatures, with array `i` corresponding to a strain rate `epsilon_p_dot[i]` and initial sample temperature `T_init[i]`. For high strain rates, these arrays of temperatures are calculated by the function `calc_temps` that is in module `bayes_stress_strain_utils` in Appendix C and discussed in Section 6.1. For low strain rates, the stress-strain curves are taken to be isothermal, and the temperature for all data points in the curve is the initial sample temperature.

Plots of the calculated temperatures are shown in Fig. 11. For reference, the code for generating these plots is as follows:

```
import matplotlib.pyplot as plt
import itertools

markers = ["None", "o", "v", "^", "<", ">",
           "1", "2", "3", "4", "8", "s", "p", "P",
           "*", "h", "H", "+", "x", "X", "D", "d"]

linestyles = ["solid", "dashed", "dashdot", "dotted"]

for i in range(len(epsilon_p_dot)):
    if epsilon_p_dot[i] > 1.0:

        plt.figure(figsize = (3.5, 4.0))
        marker_line_combo = itertools.product(markers, linestyles)

        for bTQ_fA in beta_TQ_f_area:
            marker, linestyle = next(marker_line_combo)

            plt.plot(epsilon_p[i], T[bTQ_fA][i],
                     linestyle = linestyle,
                     marker = marker,
                     markersize = 3,
                     markevery = int(len(epsilon_p[i])/25) + 1,
                     label = "$\\beta_{{TQ}}$ = {}, $f_{{area}}$ = {}".format(*
                         bTQ_fA))

        plt.xlabel("$\\epsilon_p$")
        plt.ylabel("Temperature (K)")
        plt.legend(loc = "upper left", labelspacing = 0.025)

        ymin, ymax = plt.ylim()
        ymax += 0.4*(ymax - ymin)
        plt.ylim(ymax = ymax)

        plt.tight_layout()
```

```
    out_file = "temps_for_T_init{}K_edot{}_per_s.pdf".format(
                        T_init_str[i], epsilon_p_dot_str[i])

    plt.savefig(os.path.join("plot_files", out_file))
```



(a) 77 K, 2500/s (b) 298 K, 3500/s (c) 298 K, 7000/s

(c) 473 K, 3000/s (d) 673 K, 3000/s (e) 873 K, 3500/s

Fig. 11 Temperatures as estimated in Python along stress-strain curves with the initial temperatures and strain rates shown, given the values of β_{TQ} and f_{area} in Table 1

The Gzip-compressed[29] Python pickle files to be used in fitting the Johnson-Cook model are saved as shown. These files include both the MIDAS data and the parameters for the priors of the Johnson-Cook model, i.e. $A_{guess\ mean}$, and so on, that are read in from JSON files:

```
JC_priors = bssu.read_from_json_file(
    os.path.join(os.pardir, "Other_data", "JC_priors.json")
)

JC_other_data = bssu.read_from_json_file(
    os.path.join(os.pardir, "Other_data", "JC_other_data.json")
)

T_room = JC_other_data["T_room"]
list_inds_JC = [i for i in range(len(T_init)) if T_init[i] >= T_room]
```

```
dict_for_pystan_JC = {
    "num_curves": len(list_inds_JC),
    "curve_sizes": [len(sigma[i]) for i in list_inds_JC],
    "epsilon_p_dot": [epsilon_p_dot[i] for i in list_inds_JC],
    "epsilon_p": np.concatenate([epsilon_p[i] for i in list_inds_JC]),
    "sigma": np.concatenate([sigma[i] for i in list_inds_JC])
}

dict_for_pystan_JC.update(JC_priors)
dict_for_pystan_JC.update(JC_other_data)

bssu.save_to_pickle_file(dict_for_pystan_JC,
                         os.path.join("pkl_data_files",
                                      "Main_data_for_JC_pystan.pkl.gz"))

dict_for_pymc3_JC = {
    "epsilon_p_dot": [epsilon_p_dot[i] for i in list_inds_JC],
    "epsilon_p": [epsilon_p[i] for i in list_inds_JC],
    "sigma": [sigma[i] for i in list_inds_JC],
    "prior_params": JC_priors
}

dict_for_pymc3_JC.update(JC_other_data)

bssu.save_to_pickle_file(dict_for_pymc3_JC,
                         os.path.join("pkl_data_files",
                                      "Main_data_for_JC_pymc3.pkl.gz"))

for bTQ_fA in beta_TQ_f_area:
    bTQ, fA = bTQ_fA
    bssu.save_to_pickle_file(
        [T[bTQ_fA][i] for i in list_inds_JC],
        os.path.join("pkl_data_files",
                     "T_beta{}_farea{}_JC.pkl.gz".format(bTQ,fA)))
```

Despite appearances, the previous Python code is still fairly similar to the code used to save the simulated data for testing the Johnson-Cook model, but there are of course some significant differences:

- The use of `list_inds_JC`, which stores the value of each index `i` for which `T_init[i]` is greater than or equal to `T_room`. These indices are used to select the components of lists that correspond to initial temperatures no less than T_{room}, since the Johnson-Cook model cannot be used for such temperatures.
- Whereas the pickle files for the simulated data included the temperatures for points along the stress-strain curve, here the calculated temperatures are saved to separate pickle files. The reason for this is that different fits are

to be done for the different combinations of β_{TQ} and f_{area} in Table 1, so each fit combines data from `Main_data_for_JC_pystan.pkl.gz` or `Main_data_for_JC_pymc3.pkl.gz` and the pickle file that corresponds to temperatures calculated for a particular combination of β_{TQ} and f_{area}.

Similarly, Gzip-compressed[29] pickle files to be used in fitting the Zerilli-Armstrong (BCC) model are saved as follows. Since the Zerilli-Armstrong model can accept any absolute temperature, there is no need for a list analogous to `list_inds_JC`.

```
ZA_BCC_priors = bssu.read_from_json_file(
    os.path.join(os.pardir, "Other_data", "ZA_BCC_priors.json")
)

dict_for_pystan_ZA_BCC = {
    "num_curves": len(epsilon_p_dot),
    "curve_sizes": [len(s) for s in sigma],
    "epsilon_p_dot": epsilon_p_dot,
    "epsilon_p": np.concatenate(epsilon_p),
    "sigma": np.concatenate(sigma)
}

dict_for_pystan_ZA_BCC.update(ZA_BCC_priors)

bssu.save_to_pickle_file(dict_for_pystan_ZA_BCC,
                         os.path.join("pkl_data_files",
                                      "Main_data_for_ZA_BCC_pystan.pkl.gz"))

bssu.save_to_pickle_file({
        "epsilon_p_dot": epsilon_p_dot,
        "epsilon_p": epsilon_p,
        "sigma": sigma,
        "prior_params": ZA_BCC_priors
    },
    os.path.join("pkl_data_files",
                 "Main_data_for_ZA_BCC_pymc3.pkl.gz"))

for bTQ_fA in beta_TQ_f_area:
    bTQ, fA = bTQ_fA
    bssu.save_to_pickle_file(
        T[bTQ_fA],
        os.path.join("pkl_data_files",
                     "T_beta{}_farea{}_ZA_BCC.pkl.gz".format(bTQ,fA)))
```

6.3 Fitting Johnson-Cook Model to Experimental Data with PyStan

After importing the modules `os`, `numpy`, and `bayes_stress_strain_utils`, the needed data for the $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ case, along with the Johnson-Cook PyStan model that has been saved to an pickle file in Section 5.3, are read in as follows. The function `np.concatenate` is used to concatenate all the 1-D arrays in the list of arrays stored in `T_beta0.9_farea0.75_JC.pkl.gz`.

```
import os
import numpy as np
import bayes_stress_strain_utils as bssu

my_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files", "Main_data_for_JC_pystan.pkl.gz"))

my_data["T"] = np.concatenate(bssu.read_from_pickle_file(
    os.path.join("pkl_data_files","T_beta0.9_farea0.75_JC.pkl.gz")))

jc_model = bssu.read_from_pickle_file(os.path.join("compiled_stan_models",
                                                   "jc.pkl"))
```

At this point, MCMC is about to be attempted. For the sake of reproducibility, the seed for random number generation is set. Again, the functionality of Python's `time` module is used to estimate the elapsed time in seconds.

```
import time

start_time = time.perf_counter()
jc_fit = jc_model.sampling(data = my_data, seed = 12345)
elapsed_time = time.perf_counter() - start_time

bssu.print_stan_summary(jc_fit)
print('Elapsed time: {} s'.format(elapsed_time))
```

The output from this, including both summary statistics and elapsed time, is as follows:

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/pystan/misc.py:399:
FutureWarning: Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  elif np.issubdtype(np.asarray(v).dtype, float):


0.0 of 4000 iterations ended with a divergence (0.0%)
623 of 4000 iterations saturated the maximum tree depth of 10 (15.575%)
  Run again with max_depth set to a larger value to avoid saturation
E-BFMI indicated no pathological behavior
Inference for Stan model: anon_model_5ba595b82fabba44dd5db1896f739604.
```

```
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

              mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
A           573.94    1.62  54.93 454.61 538.92 577.77  611.7 674.68   1148    1.0
B           985.07    1.54  52.35 890.03 948.63 981.15 1018.4 1100.0   1150    1.0
n             0.08  1.7e-4 5.8e-3   0.07   0.07   0.08   0.08   0.09   1193    1.0
C           4.5e-3  1.9e-6 7.9e-5 4.4e-3 4.5e-3 4.5e-3 4.6e-3 4.7e-3   1716    1.0
m             1.05  7.7e-5 3.6e-3   1.04   1.05   1.05   1.05   1.06   2179    1.0
sd_sigma[0]   9.38  8.0e-3   0.35   8.72   9.13   9.36    9.6  10.12   1951    1.0
sd_sigma[1]  32.58    0.01   0.67  31.32  32.14  32.57  33.02  33.95   2439    1.0
lp__         -6807    0.05    1.9  -6811  -6808  -6806  -6805  -6804   1443    1.0

Samples were drawn using NUTS at Fri May 18 09:23:00 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Elapsed time: 790.2527485881001 s
```

Here, the MCMC ran significantly longer than it did in the testing run in Section 5.3, about 10 to 13 min. The potential scale reduction factors (i.e., `Rhat`) look reasonable, but there are warnings about tree depth, so the MCMC is to be run again with `max_treedepth` set to a higher value.

```
start_time = time.perf_counter()
jc_fit = jc_model.sampling(data = my_data, seed = 12345,
                           control = {'max_treedepth': 15})
elapsed_time = time.perf_counter() - start_time

bssu.print_stan_summary(jc_fit)
print('Elapsed time: {} s'.format(elapsed_time))
```

The following is the output for this new MCMC run:

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/pystan/misc.py:399:
FutureWarning: Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  elif np.issubdtype(np.asarray(v).dtype, float):


0.0 of 4000 iterations ended with a divergence (0.0%)
0 of 4000 iterations saturated the maximum tree depth of 15 (0.0%)
E-BFMI indicated no pathological behavior
Inference for Stan model: anon_model_5ba595b82fabba44dd5db1896f739604.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

              mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
A           574.95    1.82  52.91 463.43 541.17  578.0 610.74 673.86    843    1.0
```

```
B           984.02    1.73  50.48 890.31 949.61 980.97 1015.9 1091.0    848    1.0
n             0.08  1.9e-4 5.6e-3   0.07   0.07   0.08   0.08   0.09    898    1.0
C           4.5e-3  2.0e-6 8.0e-5 4.4e-3 4.5e-3 4.5e-3 4.6e-3 4.7e-3   1667    1.0
m             1.05  8.0e-5 3.6e-3   1.04   1.05   1.05   1.05   1.06   2058    1.0
sd_sigma[0]   9.36  7.9e-3   0.35   8.68   9.12   9.36    9.6  10.08   2026    1.0
sd_sigma[1]  32.59    0.01   0.69  31.26  32.13  32.59  33.05  33.97   2279    1.0
lp__         -6807    0.05   1.93  -6811  -6808  -6806  -6805  -6804   1274    1.0

Samples were drawn using NUTS at Fri May 18 09:45:20 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Elapsed time: 1032.850906773936 s
```

This output shows no further warnings, and both the effective sample sizes and potential scale reduction factors still look reasonable. However, the mean value of A, which is supposed to approximate the yield stress,[7] appears slightly low for RHA. Nonetheless, the samples and summary from the MCMC run are saved for future examination, using the function `save_stan_fit_to_csv` from the Python module `bayes_stress_strain_utils` in Appendix C. To save disk space, the samples are saved to a Gzip-compressed[29] CSV file:

```
bssu.save_stan_fit_to_csv(
    jc_fit,
    os.path.join("summaries",
                 "jc_MIDAS_pystan_summary_weak_prior_bTQ09_fA075.csv"),
    os.path.join("samples",
                 "jc_MIDAS_pystan_samples_weak_prior_bTQ09_fA075.csv.gz"))
```

The string "`weak_prior`" in the names of the CSV files indicates that these MCMC results are obtained with weakly informative priors. The string "`bTQ09`" indicates that β_{TQ} is 0.9 (with "`bTQ`" referring to β_{TQ} and "`09`" referring to 0.9), while "`fA075`" indicates that f_{area} ("`fA`") is 0.75 ("`075`").

At this point, an MCMC run is about to be run with the strongly informative prior for A. If one starts from the same Python session that has been used for the MCMC runs with the weak prior, then only a small change to the `my_data` variable is needed:

```
new_RHA_priors = bssu.read_from_json_file(
    os.path.join(os.pardir, "Other_data",
                 "JC_prior_A_Benck.json"))

my_data["A_guess_mean"] = new_RHA_priors["A_guess_mean"]
my_data["A_guess_sd"] = new_RHA_priors["A_guess_sd"]
```

MCMC is then run just as before, with the same `seed` and `max_treedepth` values:

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/pystan/misc.py:399:
FutureWarning: Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  elif np.issubdtype(np.asarray(v).dtype, float):


0.0 of 4000 iterations ended with a divergence (0.0%)
0 of 4000 iterations saturated the maximum tree depth of 15 (0.0%)
E-BFMI indicated no pathological behavior
Inference for Stan model: anon_model_5ba595b82fabba44dd5db1896f739604.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

               mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
A            700.53    0.29  10.29 680.02 693.63 700.61 707.66  720.5   1295    1.0
B            865.59    0.27   9.74 846.72 858.79 865.48 872.13 885.05   1345    1.0
n              0.09  4.7e-5 1.7e-3   0.09   0.09   0.09   0.09    0.1   1393    1.0
C             4.5e-3  1.7e-6 8.0e-5 4.4e-3 4.5e-3 4.5e-3 4.6e-3 4.7e-3   2316    1.0
m              1.05  7.0e-5 3.6e-3   1.04   1.04   1.05   1.05   1.05   2655    1.0
sd_sigma[0]    9.65  7.4e-3   0.35   9.01   9.41   9.65   9.88  10.38   2225    1.0
sd_sigma[1]   32.33    0.01   0.65  31.12  31.88  32.31  32.78  33.63   2659    1.0
lp__          -6810    0.05   1.84  -6814  -6811  -6809  -6808  -6807   1485    1.0

Samples were drawn using NUTS at Wed May 30 14:11:57 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Elapsed time: 398.3065286980709 s
```

In this output, both the effective sample sizes and potential scale reduction factors still look reasonable, and at this point, the mean of parameter A looks reasonable as well. The elapsed time is shorter as well. At this point, the results need to be saved for further analysis. The samples and summary from MCMC may be saved as follows:

```
bssu.save_stan_fit_to_csv(
    jc_fit,
    os.path.join("summaries",
                 "jc_MIDAS_pystan_summary_strong_prior_on_A_bTQ09_fA075.csv"),
    os.path.join("samples",
                 "jc_MIDAS_pystan_samples_strong_prior_on_A_bTQ09_fA075.csv.gz"))
```

Here, the string "`strong_prior_on_A`" indicates that a strongly informative prior is used for A.

Fits for the Johnson-Cook model have been done for the rest of the combinations of β_{TQ} and f_{area} in Table 1, for both strong and weak priors. Loading of the data for these fits proceeds much as before, with `T_beta0.9_farea0.75_JC.pkl.gz` replaced with the file for a different pair of β_{TQ} and f_{area}, such as `T_beta0.6_farea0.95_JC.pkl.gz` for $\beta_{TQ} = 0.6$ and $f_{area} = 0.95$. The means and standard deviations of the resulting fitted parameters are in Ramsey.[4]

6.4 Fitting Johnson-Cook Model to Experimental Data with PyMC3

The needed temperature data calculated with $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ are to be read in after importing the Python modules `os`, `numpy`, `pymc3`, `jc_pymc3`, and `bayes_stress_strain_utils`, and then the Johnson-Cook PyMC3 model is to be instantiated with that data, as follows. Again, `os.environ["MKL_THREADING_LAYER"]` is set to the string `"GNU"` *before* importing the `pymc3` module or any module that loads `pymc3`:

```
import os

os.environ["MKL_THREADING_LAYER"] = "GNU"

import numpy as np
import pymc3 as pm
import jc_pymc3
import bayes_stress_strain_utils as bssu

my_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files", "Main_data_for_JC_pymc3.pkl.gz"))

my_data["T"] = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files","T_beta0.9_farea0.75_JC.pkl.gz"))

jc_model = jc_pymc3.make_jc_model(**my_data)
```

Since all the keys of the dictionary `my_data` are the names of the arguments to the function `make_jc_model`, one can use `**my_data` as an argument to that function instead of specifying the arguments individually as has been done in Section 5.4.

At this point, MCMC is about to be attempted. For the sake of reproducibility, the seed for random number generation is set. Because initial values are needed to get the Johnson-Cook PyMC3 model to even finish an MCMC run on simulated data, they are supplied here when running MCMC with real data. Again, the values for

the arguments `draws` and `tune` in the `sample` function of PyMC3 are set so that the number of samples used for warmup (or tuning) and final sampling in PyMC3 are the same as they are with Stan, and the values for the arguments `chains` and `cores` are chosen so that the number of chains generated in parallel is the same for both PyMC3 and Stan:

```
import traceback

prior_params = my_data["prior_params"]

start_vals = {
    'A': prior_params['A_guess_mean'],
    'B': prior_params['B_guess_mean'],
    'n': prior_params['n_alpha']/
            (prior_params['n_alpha'] + prior_params['n_beta']),
    'C': prior_params['C_guess_mean'],
    'm': prior_params['m_guess_mean'],
    'sd_sigma': np.asarray(prior_params['sd_sigma_guess_mean'])
}

try:
    with jc_model:
        jc_trace = pm.sample(draws = 1000, tune = 1000,
                             chains = 4, cores = 4,
                             random_seed = 12345,
                             start = start_vals)

    print(pm.summary(jc_trace))

except:
    err_filename = "jc_trace_fit_errs.txt"
    print("Exception encountered: see file {}!".format(err_filename))
    with open(err_filename, "w") as err_file:
        traceback.print_exc(file=err_file)
```

The following is the output for this MCMC run:

```
Auto-assigning NUTS sampler...
INFO:pymc3:Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
INFO:pymc3:Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
INFO:pymc3:Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sd_sigma, m, C, n, B, A]
INFO:pymc3:NUTS: [sd_sigma, m, C, n, B, A]
Sampling 4 chains: 100%|[redacted]| 8000/8000 [09:47<00:00,  3.65draws/s]
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/mkl_fft/_numpy_fft.py:1044:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
array index, `arr[np.array(seq)]`, which will result either in an error or a different
result.
```

```
  output = mkl_fft.rfftn_numpy(a, s, axes)
```

```
                   mean         sd  mc_error     hpd_2.5     hpd_97.5  \
A            574.837970  57.121351  1.675024  461.590354   681.499341
B            984.264375  54.502762  1.598589  882.271991  1092.301880
n              0.077246   0.005981  0.000172    0.065751     0.089080
C              0.004518   0.000078  0.000001    0.004365     0.004671
m              1.048044   0.003640  0.000071    1.040626     1.055227
sd_sigma__0    9.381372   0.359893  0.007247    8.694195    10.099319
sd_sigma__1   32.559318   0.654547  0.012048   31.324634    33.878966

                   n_eff      Rhat
A            1064.317830  1.001727
B            1066.477129  1.001703
n            1131.223756  1.001803
C            2096.939128  1.000209
m            2184.229497  0.999901
sd_sigma__0  2039.472649  1.000540
sd_sigma__1  2525.710209  0.999642
```

In this output, the diagnostics `n_eff` and `Rhat` look reasonable. The sampling time is also of the same order of magnitude as the MCMC sampling done for PyStan in Section 6.3. As with the corresponding PyStan run, the value of A, which is approximately the yield stress,[7] appears slightly low for RHA. Nonetheless, the samples and summary from the MCMC run are saved for future examination, using the convenience function `save_pymc3_trace_to_csv` from the Python module `bayes_stress_strain_utils` in Appendix C. To save disk space, the samples are saved to a Gzip-compressed[29] CSV file:

```
bssu.save_pymc3_trace_to_csv(
    jc_trace,
    os.path.join("summaries",
                 "jc_MIDAS_pymc3_summary_weak_prior_bTQ09_fA075.csv"),
    os.path.join("samples",
                 "jc_MIDAS_pymc3_samples_weak_prior_bTQ09_fA075.csv.gz"))
```

If the directories `summaries` and `samples` do not yet exist, the function `save_pymc3_trace_to_csv` creates them. The string "`weak_prior`" in the names of the CSV files indicates that these MCMC results have been obtained with weakly informative priors. The string "`bTQ09`" indicates that β_{TQ} is 0.9 (with "`bTQ`" referring to β_{TQ} and "`09`" referring to 0.9), while "`fA075`" indicates that f_{area} ("`fA`") is 0.75 ("`075`").

At this point, an MCMC run is about to be run with the strongly informative prior for A. If one starts from the same Python session used for the MCMC runs with

the weak prior, then only a small change to the `my_data` variable is needed. For consistency with the new priors, the initial value for parameter A is changed as well:

```
new_RHA_priors = bssu.read_from_json_file(
    os.path.join(os.pardir, "Other_data", "JC_prior_A_Benck.json"))

my_data["prior_params"]["A_guess_mean"] = new_RHA_priors["A_guess_mean"]
my_data["prior_params"]["A_guess_sd"] = new_RHA_priors["A_guess_sd"]

start_vals["A"] = new_RHA_priors["A_guess_mean"]
```

After instantiating the PyMC3 model with the new priors, MCMC is then run just as before, and the output from this is as follows:

```
Auto-assigning NUTS sampler...
INFO:pymc3:Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
INFO:pymc3:Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
INFO:pymc3:Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sd_sigma, m, C, n, B, A]
INFO:pymc3:NUTS: [sd_sigma, m, C, n, B, A]
Sampling 4 chains: 100%|[redacted]| 8000/8000 [02:02<00:00, 65.14draws/s]
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/mkl_fft/_numpy_fft.py:1044:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
array index, `arr[np.array(seq)]`, which will result either in an error or a different
result.
  output = mkl_fft.rfftn_numpy(a, s, axes)
```

	mean	sd	mc_error	hpd_2.5	hpd_97.5	\
A	700.618008	10.111129	0.254464	680.685215	720.184043	
B	865.515315	9.595868	0.242256	846.173374	883.677783	
n	0.092820	0.001712	0.000039	0.089487	0.096181	
C	0.004541	0.000078	0.000002	0.004387	0.004689	
m	1.047172	0.003566	0.000078	1.040496	1.054614	
sd_sigma__0	9.652956	0.352611	0.006741	8.982017	10.349625	
sd_sigma__1	32.350212	0.646231	0.012064	31.091673	33.667343	

	n_eff	Rhat
A	1398.137721	1.001999
B	1435.118621	1.001724
n	1535.714358	1.002184
C	1789.191987	1.000755
m	1809.571121	1.000885
sd_sigma__0	2456.495922	1.000672
sd_sigma__1	2742.718991	1.000073

In this output, both the effective sample sizes and potential scale reduction factors still look reasonable, and at this point, the mean of parameter A looks reasonable

as well. The elapsed time is shorter as well. At this point, the results need to be saved for further analysis. The samples and summary from MCMC may be saved as follows:

```
bssu.save_pymc3_trace_to_csv(
    jc_trace,
    os.path.join("summaries",
                 "jc_MIDAS_pymc3_summary_strong_prior_on_A_bTQ09_fA075.csv"),
    os.path.join("samples",
                 "jc_MIDAS_pymc3_samples_strong_prior_on_A_bTQ09_fA075.csv.gz"))
```

Here, the string "`strong_prior_on_A`" indicates that a strongly informative prior is used for A.

Fits for the Johnson-Cook model have been done for the rest of the combinations of β_{TQ} and f_{area} in Table 1, for both strong and weak priors. Loading of the data for these fits proceeds much as before, with `T_beta0.9_farea0.75_JC.pkl.gz` replaced with the file for a different pair of β_{TQ} and f_{area}, such as `T_beta0.6_farea0.95_JC.pkl.gz` for $\beta_{TQ} = 0.6$ and $f_{area} = 0.95$. The means and standard deviations of the resulting fitted parameters are in Ramsey.[4]

6.5 Fitting Zerilli-Armstrong (BCC) Model to Experimental Data with PyStan

Much as with the Johnson-Cook model, after importing the modules `os`, `numpy`, and `bayes_stress_strain_utils`, the needed data for the $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ case, along with the Zerilli-Armstrong (BCC) PyStan model that has been saved to an pickle file in Section 5.6, are read in as follows:

```
import os
import numpy as np
import bayes_stress_strain_utils as bssu

my_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files", "Main_data_for_ZA_BCC_pystan.pkl.gz"))

# Using np.concatenate to turn a list of 1-dimensional arrays into
# one long 1-dimensional array
my_data["T"] = np.concatenate(bssu.read_from_pickle_file(
    os.path.join("pkl_data_files","T_beta0.9_farea0.75_ZA_BCC.pkl.gz")))

za_bcc_model = bssu.read_from_pickle_file(os.path.join("compiled_stan_models",
                                                       "za_bcc.pkl"))
```

As the testing of the Zerilli-Armstrong model showed, initial values for the model parameters are needed as well, and these are set to the means of the priors:

```
init_vals = {
    "C0": my_data["C0_guess_mean"],
    "C1": my_data["C1_guess_mean"],
    "C3": my_data["C3_guess_mean"],
    "C4": my_data["C4_guess_mean"],
    "C5": my_data["C5_guess_mean"],
    "n": my_data["n_alpha"]/(my_data["n_alpha"] + my_data["n_beta"])
}
```

The Zerilli-Armstrong PyStan model that has been saved to a pickle file is to be loaded, and then MCMC is to be attempted. For the sake of reproducibility, the seed for random number generation is set. Since the only way PyStan prints elapsed time is via terminal output (which may not be visible in, for example, a Jupyter notebook), the functionality of Python's `time` module is used to estimate the elapsed time in seconds. Also, this time, the initial values are set with a `lambda` expression rather than a list of dictionaries.

```
import time

start_time = time.perf_counter()
za_bcc_fit = za_bcc_model.sampling(data = my_data,
                                   init = (lambda : init_vals),
                                   seed = 12345)
elapsed_time = time.perf_counter() - start_time

bssu.print_stan_summary(za_bcc_fit)
print('Elapsed time: {} s'.format(elapsed_time))
```

The output from the previous code is shown, and the diagnostics in this output are reasonable:

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/pystan/misc.py:399:
FutureWarning: Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  elif np.issubdtype(np.asarray(v).dtype, float):


0.0 of 4000 iterations ended with a divergence (0.0%)
0 of 4000 iterations saturated the maximum tree depth of 10 (0.0%)
E-BFMI indicated no pathological behavior
Inference for Stan model: anon_model_94ebba402ea6796923b7e03a53b54c34.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.
```

```
               mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
C0           108.55    0.73   25.9  56.45  91.08  109.4 126.83 156.62   1273    1.0
C1           1529.4    0.18    8.6 1512.9 1523.5 1529.2 1535.1 1546.8   2173    1.0
C3           2.2e-3  7.4e-7 3.1e-5 2.1e-3 2.2e-3 2.2e-3 2.2e-3 2.2e-3   1721    1.0
C4           4.1e-5  1.6e-8 7.3e-7 4.0e-5 4.1e-5 4.1e-5 4.2e-5 4.3e-5   2182    1.0
C5           748.58    0.61   22.1 706.88 732.99 748.11 763.36 793.84   1308    1.0
n              0.16  2.6e-4 9.7e-3   0.14   0.15   0.16   0.16   0.18   1426    1.0
sd_sigma[0]   31.62    0.02   0.94  29.88  30.97  31.58  32.23  33.58   2914    1.0
sd_sigma[1]   47.47    0.02   0.84  45.85   46.9  47.45  48.02  49.17   3109    1.0
lp__          -9766    0.06   2.07  -9770  -9767  -9765  -9764  -9763   1392    1.0

Samples were drawn using NUTS at Mon May 21 12:55:39 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Elapsed time: 155.38615178316832 s
```

The samples and summary from the MCMC run are saved for future examination via the same `save_stan_fit_to_csv` function used with the results of the Johnson-Cook model:

```
bssu.save_stan_fit_to_csv(
    za_bcc_fit,
    os.path.join("summaries",
                 "za_bcc_MIDAS_pystan_summary_bTQ09_fA075.csv"),
    os.path.join("samples",
                 "za_bcc_MIDAS_pystan_samples_bTQ09_fA075.csv.gz"))
```

In the previous fit for the Zerilli-Armstrong model, one may have observed that the $SD_{\sigma,1}$ is only slightly less than $SD_{\sigma,2}$, whereas with the Johnson-Cook model, the former is about three times less than the latter. To see if this is due to the Zerilli-Armstrong model being fit to data not used in the fit for the Johnson-Cook model, a new fit is done, using only the data used in the latter fit. If one starts from the same Python session used for the previous fit, then an MCMC run with the new data can be done as shown:

```
JC_main_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files", "Main_data_for_JC_pystan.pkl.gz"))

my_data["num_curves"] = JC_main_data["num_curves"]
my_data["curve_sizes"] = JC_main_data["curve_sizes"]
my_data["epsilon_p_dot"] = JC_main_data["epsilon_p_dot"]
my_data["epsilon_p"] = JC_main_data["epsilon_p"]
my_data["sigma"] = JC_main_data["sigma"]

my_data["T"] = np.concatenate(bssu.read_from_pickle_file(
    os.path.join("pkl_data_files","T_beta0.9_farea0.75_JC.pkl.gz")))
```

```
start_time = time.perf_counter()
za_bcc_fit = za_bcc_model.sampling(data = my_data,
                                   init = (lambda : init_vals),
                                   seed = 12345)
elapsed_time = time.perf_counter() - start_time

bssu.print_stan_summary(za_bcc_fit)
print('Elapsed time: {} s'.format(elapsed_time))
```

The following are the results from the MCMC run:

```
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/pystan/misc.py:399:
FutureWarning: Conversion of the second argument of issubdtype from `float` to
`np.floating` is deprecated. In future, it will be treated as `np.float64 ==
np.dtype(float).type`.
  elif np.issubdtype(np.asarray(v).dtype, float):


0.0 of 4000 iterations ended with a divergence (0.0%)
0 of 4000 iterations saturated the maximum tree depth of 10 (0.0%)
E-BFMI indicated no pathological behavior
Inference for Stan model: anon_model_94ebba402ea6796923b7e03a53b54c34.
4 chains, each with iter=2000; warmup=1000; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=4000.

               mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
C0             1.84    0.03   1.82   0.04    0.5    1.3   2.58   6.77   3272    1.0
C1           1535.4    0.31  12.28 1510.7 1527.1 1535.6 1543.7 1559.6   1525    1.0
C3          1.4e-3  6.1e-7 2.3e-5 1.4e-3 1.4e-3 1.4e-3 1.4e-3 1.4e-3   1406    1.0
C4          2.6e-5  1.2e-8 5.3e-7 2.5e-5 2.6e-5 2.6e-5 2.7e-5 2.8e-5   1957    1.0
C5           590.8    0.27  10.46 570.62 583.76 590.67 597.72  611.6   1472    1.0
n             0.18  2.0e-4 7.6e-3   0.16   0.17   0.18   0.18   0.19   1365    1.0
sd_sigma[0]  13.92    0.01   0.58  12.84  13.52  13.91  14.31  15.11   2278    1.0
sd_sigma[1]  43.25    0.02    0.9  41.48  42.63  43.24  43.86  45.07   2243    1.0
lp__         -7348    0.05   2.01  -7353  -7349  -7348  -7347  -7345   1465    1.0

Samples were drawn using NUTS at Tue May 22 10:49:54 2018.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
Elapsed time: 89.19370407611132 s
```

The relationship of $SD_{\sigma,1}$ to $SD_{\sigma,2}$ is now similar to what it is for the Johnson-Cook model. The samples and summary from this MCMC run are also saved for future examination as follows:

```
bssu.save_stan_fit_to_csv(
    za_bcc_fit,
    os.path.join("summaries",
                 "za_bcc_MIDAS_pystan_summary_JC_data_bTQ09_fA075.csv"),
```

```
    os.path.join("samples",
                 "za_bcc_MIDAS_pystan_samples_JC_data_bTQ09_fA075.csv.gz"))
```

Fits for the Zerilli-Armstrong (BCC) model have been done for the rest of the combinations of β_{TQ} and f_{area} in Table 1, for both the case where all MIDAS data are included and the case where only the MIDAS data used to fit the Johnson-Cook model are included. The means and standard deviations of the resulting fitted parameters are in Ramsey.[4]

6.6 Fitting Zerilli-Armstrong (BCC) Model to Experimental Data with PyMC3

Much as with the Johnson-Cook model, the needed data for the $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ case are read in after importing the modules `os`, `numpy`, `pymc3`, `za_bcc_pymc3`, and `bayes_stress_strain_utils`, and then the Zerilli-Armstrong (BCC) PyMC3 model is instantiated with that data as follows. Again, `os.environ["MKL_THREADING_LAYER"]` is set to the string `"GNU"` *before* importing the `pymc3` module or any module that loads `pymc3`:

```
import os

os.environ["MKL_THREADING_LAYER"] = "GNU"

import numpy as np
import pymc3 as pm
import za_bcc_pymc3
import bayes_stress_strain_utils as bssu

my_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files", "Main_data_for_ZA_BCC_pymc3.pkl.gz"))

my_data["T"] = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files","T_beta0.9_farea0.75_ZA_BCC.pkl.gz"))

za_bcc_model = za_bcc_pymc3.make_za_bcc_model(**my_data)
```

As the testing of the Zerilli-Armstrong PyMC3 model showed, initial values for the model parameters are needed as well, and these are set to the means of the priors:

```
prior_params = my_data["prior_params"]

start_vals = {
    "C0": prior_params["C0_guess_mean"],
    "C1": prior_params["C1_guess_mean"],
    "C3": prior_params["C3_guess_mean"],
    "C4": prior_params["C4_guess_mean"],
```

```
    "C5": prior_params["C5_guess_mean"],
    "n": prior_params["n_alpha"]/
        (prior_params["n_alpha"] + prior_params["n_beta"]),
    "sd_sigma": prior_params['sd_sigma_guess_mean']
}
```

At this point, MCMC is about to be attempted. For the sake of reproducibility, the seed for random number generation is set. Again, the values for the arguments `draws` and `tune` in the `sample` function of PyMC3 are set so that the number of samples used for warmup (or tuning) and final sampling in PyMC3 are the same as they are in Stan, and the values for the arguments `chains` and `cores` are chosen so that the number of chains generated in parallel is the same for both PyMC3 and Stan:

```
import traceback

try:
    with za_bcc_model:
        za_bcc_trace = pm.sample(draws = 1000, tune = 1000,
                                 chains = 4, cores = 4,
                                 random_seed = 12345,
                                 start = start_vals)

    print(pm.summary(za_bcc_trace))
except:
    err_filename = "za_bcc_trace_fit_errs.txt"
    print("Exception encountered: see file {}!".format(err_filename))
    with open(err_filename, "w") as err_file:
        traceback.print_exc(file=err_file)
```

The output from this is as follows:

```
Auto-assigning NUTS sampler...
INFO:pymc3:Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
INFO:pymc3:Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
INFO:pymc3:Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
INFO:pymc3:NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
Sampling 4 chains: 100%|[redacted]| 8000/8000 [02:43<00:00, 15.44draws/s]
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/mkl_fft/_numpy_fft.py:1044:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
array index, `arr[np.array(seq)]`, which will result either in an error or a different
result.
  output = mkl_fft.rfftn_numpy(a, s, axes)
There was 1 divergence after tuning. Increase `target_accept` or reparameterize.
ERROR:pymc3:There was 1 divergence after tuning. Increase `target_accept` or
```

```
reparameterize.
There were 3 divergences after tuning. Increase 'target_accept' or reparameterize.
ERROR:pymc3:There were 3 divergences after tuning. Increase 'target_accept' or
reparameterize.
There were 23 divergences after tuning. Increase 'target_accept' or reparameterize.
ERROR:pymc3:There were 23 divergences after tuning. Increase 'target_accept' or
reparameterize.

                    mean             sd      mc_error      hpd_2.5  \
C0            108.247409  2.619048e+01  6.798457e-01    59.346535
C1           1529.098273  8.649828e+00  1.997850e-01  1510.450806
C3              0.002191  3.045333e-05  7.229348e-07     0.002134
C4              0.000041  7.138980e-07  1.280137e-08     0.000040
C5            749.526065  2.223788e+01  5.661143e-01   706.786599
n               0.158223  9.669127e-03  2.454245e-04     0.140425
sd_sigma__0    31.626514  9.437332e-01  1.755603e-02    29.872822
sd_sigma__1    47.499352  8.430675e-01  1.489903e-02    45.740558

                hpd_97.5        n_eff      Rhat
C0            160.837920  1504.541261  0.999996
C1           1544.914149  1750.421236  1.000890
C3              0.002251  1660.081945  1.001599
C4              0.000042  2572.793796  1.000066
C5            792.024376  1474.481134  1.000598
n               0.177682  1541.147509  1.000632
sd_sigma__0    33.551697  2948.766445  1.000009
sd_sigma__1    49.079264  3151.816049  1.000650
```

There are warning about divergences, so MCMC is rerun with a larger `target_accept` value:

```
try:
    with za_bcc_model:
        za_bcc_trace = pm.sample(draws = 1000, tune = 1000,
                                 chains = 4, cores = 4,
                                 random_seed = 12345,
                                 nuts_kwargs = {"target_accept": 0.9},
                                 start = start_vals)

    print(pm.summary(za_bcc_trace))
except:
    err_filename = "za_bcc_trace_fit_errs2.txt"
    print("Exception encountered: see file {}!".format(err_filename))
    with open(err_filename, "w") as err_file:
        traceback.print_exc(file=err_file)
```

The output from this new MCMC run is shown. There are no more warnings from PyMC3 in this output, and the `n_eff` and `Rhat` diagnostics look reasonable:

```
Auto-assigning NUTS sampler...
INFO:pymc3:Auto-assigning NUTS sampler...
```

```
Initializing NUTS using jitter+adapt_diag...
INFO:pymc3:Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
INFO:pymc3:Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
INFO:pymc3:NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
Sampling 4 chains: 100%|[redacted]| 8000/8000 [03:20<00:00, 12.81draws/s]
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/mkl_fft/_numpy_fft.py:1044:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
array index, `arr[np.array(seq)]`, which will result either in an error or a different
result.
  output = mkl_fft.rfftn_numpy(a, s, axes)
```

	mean	sd	mc_error	hpd_2.5	\
C0	107.537860	2.640884e+01	7.225838e-01	54.563983	
C1	1529.125901	8.838833e+00	2.059382e-01	1511.795090	
C3	0.002190	3.056559e-05	7.485135e-07	0.002129	
C4	0.000041	7.173988e-07	1.463182e-08	0.000040	
C5	749.893994	2.271279e+01	6.210657e-01	706.512010	
n	0.157974	9.888586e-03	2.703643e-04	0.140280	
sd_sigma__0	31.634860	9.521057e-01	1.917492e-02	29.737175	
sd_sigma__1	47.489870	8.287531e-01	1.470518e-02	45.829694	

	hpd_97.5	n_eff	Rhat
C0	157.094955	1118.360428	1.000544
C1	1546.164206	1815.062160	1.000414
C3	0.002249	1624.639811	1.000653
C4	0.000042	2163.037153	1.000013
C5	794.506894	1189.976219	0.999856
n	0.178607	1162.491848	0.999985
sd_sigma__0	33.463233	2874.527410	1.000352
sd_sigma__1	49.056928	3160.862301	1.000073

The samples and summary from the MCMC run are saved for future examination as follows, via the same `save_pymc3_trace_to_csv` function used with the results of the Johnson-Cook model:

```
bssu.save_pymc3_trace_to_csv(
    za_bcc_trace,
    os.path.join("summaries",
                 "za_bcc_MIDAS_pymc3_summary_bTQ09_fA075.csv"),
    os.path.join("samples",
                 "za_bcc_MIDAS_pymc3_samples_bTQ09_fA075.csv.gz"))
```

In the previous fit for the Zerilli-Armstrong model, one may have observed that the $SD_{\sigma,1}$ is only slightly less than $SD_{\sigma,2}$, whereas with the Johnson-Cook model, the former is about three times less than the latter. To see if this is due to the Zerilli-Armstrong model being fit to data not used in the fit for the Johnson-Cook model, a

new fit is done, using only the data used in the latter fit. If one starts from the same Python session used for the previous fit, then an MCMC run with the new data can be done as shown:

```
JC_main_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files", "Main_data_for_JC_pymc3.pkl.gz"))

my_data["epsilon_p_dot"] = JC_main_data["epsilon_p_dot"]
my_data["epsilon_p"] = JC_main_data["epsilon_p"]
my_data["sigma"] = JC_main_data["sigma"]

my_data["T"] = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files","T_beta0.9_farea0.75_JC.pkl.gz"))

za_bcc_model = za_bcc_pymc3.make_za_bcc_model(**my_data)

try:
    with za_bcc_model:
        za_bcc_trace = pm.sample(draws = 1000, tune = 1000,
                                 chains = 4, cores = 4,
                                 random_seed = 12345,
                                 nuts_kwargs = {"target_accept": 0.9},
                                 start = start_vals)

    print(pm.summary(za_bcc_trace))
except:
    err_filename = "za_bcc_trace_fit_errs_jc_data.txt"
    print("Exception encountered: see file {}!".format(err_filename))
    with open(err_filename, "w") as err_file:
        traceback.print_exc(file=err_file)
```

The following is the output from the new MCMC run:

```
INFO:pymc3:Auto-assigning NUTS sampler...
INFO:pymc3:Initializing NUTS using jitter+adapt_diag...
INFO:pymc3:Multiprocess sampling (4 chains in 4 jobs)
INFO:pymc3:NUTS: [sd_sigma, n, C5, C4, C3, C1, C0]
Sampling 4 chains: 100%|[redacted]| 8000/8000 [01:49<00:00, 72.92draws/s]
/home/jjramsey/.conda/envs/Bayes/lib/python3.6/site-packages/mkl_fft/_numpy_fft.py:1044:
FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use
`arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an
array index, `arr[np.array(seq)]`, which will result either in an error or a different
result.
  output = mkl_fft.rfftn_numpy(a, s, axes)


                  mean            sd      mc_error      hpd_2.5  \
C0            1.867899  1.805520e+00  3.223922e-02     0.000012
C1         1535.414584  1.229310e+01  3.285065e-01  1511.648589
C3            0.001400  2.280600e-05  6.124216e-07     0.001354
C4            0.000026  5.479599e-07  1.215039e-08     0.000025
C5          590.788279  1.055761e+01  2.886484e-01   569.309714
```

```
n               0.178332  7.510367e-03  2.083718e-04     0.164611
sd_sigma__0    13.912941  5.962634e-01  1.391148e-02    12.746006
sd_sigma__1    43.234743  8.806380e-01  1.996328e-02    41.328536

                hpd_97.5        n_eff      Rhat
C0              5.516702  3045.875454  1.001106
C1           1559.635019  1281.619064  1.001756
C3              0.001442  1233.985375  1.001176
C4              0.000028  1749.727779  1.000019
C5            610.784355  1204.080384  1.002086
n               0.193831  1170.080314  1.001970
sd_sigma__0    15.042074  1902.934484  1.000598
sd_sigma__1    44.866560  1913.277254  1.000542
```

The relationship of $SD_{\sigma,1}$ to $SD_{\sigma,2}$ is now similar to what it is for the Johnson-Cook model. The samples and summary from this MCMC run are also saved as follows for future examination:

```
bssu.save_pymc3_trace_to_csv(
    za_bcc_trace,
    os.path.join("summaries",
                 "za_bcc_MIDAS_pymc3_summary_JC_data_bTQ09_fA075.csv"),
    os.path.join("samples",
                 "za_bcc_MIDAS_pymc3_samples_JC_data_bTQ09_fA075.csv.gz"))
```

Fits for the Zerilli-Armstrong (BCC) model have been done for the rest of the combinations of β_{TQ} and f_{area} in Table 1, for both the case where all MIDAS data are included and the case where only the MIDAS data used to fit the Johnson-Cook model are included. The means and standard deviations of the resulting fitted parameters are in Ramsey.[4]

6.7 Applying Approximate Interval Predictor Model Approach

To do the constrained optimization for the IPM to find parameter bounds for the Johnson-Cook model, one needs to load not only the function specifying the flow stress according to that model, but also its *gradient*, which here is specified via the function `jc_grad`:

```
from jc import jc
import numpy as np

def jc_grad(epsilon_p, log_epsilon_p_dot, T_star,
            A, B, n, C, m):

    dJCdA = (-T_star**m + 1)*(C*log_epsilon_p_dot + 1.0)
    dJCdB = (epsilon_p**n)*(-T_star**m + 1)*(C*log_epsilon_p_dot + 1.0)
```

```
    dJCdn = np.where(epsilon_p == 0,
                     np.full(len(epsilon_p), 0.0),
                     B*(epsilon_p**n)*(-T_star**m + 1)*
                     (C*log_epsilon_p_dot + 1.0)*
                     np.log(epsilon_p))

    dJCdC = log_epsilon_p_dot*(A + B*epsilon_p**n)*(-T_star**m + 1)

    dJCdm = np.where(T_star == 0,
                     np.full(len(T_star), 0.0),
                     -T_star**m*(A + B*epsilon_p**n)*
                     (C*log_epsilon_p_dot + 1.0)*
                     np.log(T_star))

    return np.vstack((dJCdA, dJCdB, dJCdn, dJCdC, dJCdm))
```

The derivatives in the function `jc_grad` are calculated with the aid of a symbolic computation package (in this case, SymPy[43]). However, blindly using the derivative expressions from a symbolic computation would be a problem, since the expressions for the derivatives with respect to parameters n and m are undefined where ϵ_p or T^* are zero, because of the presence of the factors $\epsilon_p^n \ln \epsilon_p$ and $(T^*)^m \ln T^*$, respectively, in those expressions. Mathematically, though, as $\epsilon_p \to 0$ and $T^* \to 0$, these factors approach zero, and the numerical calculation of the derivatives reflects that. Also, to allow the Python variables `epsilon_p` and `T_star` to be arrays, `np.where` is used rather than a raw `if` statement.

At this point, one can load in the needed data, which in this case is data prepared for PyMC3, as well as temperature data for $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$:

```
import bayes_stress_strain_utils as bssu
import os

my_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files", "Main_data_for_JC_pymc3.pkl.gz"))

my_data["T"] = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files","T_beta0.9_farea0.75_JC.pkl.gz"))

epsilon_p = my_data["epsilon_p"]

epsilon_p_dot = my_data["epsilon_p_dot"]
log_ep_dot = np.log(np.asarray(epsilon_p_dot)/
                    my_data["epsilon_p_dot_0"])

T_room = my_data["T_room"]
T_melt_minus_T_room = my_data["T_melt"] - T_room

T_star = [(T - T_room)/T_melt_minus_T_room for T in my_data["T"]]
```

To make the constrained optimization more tractable, the flow stresses are converted from units of megapascals to gigapascals:

```
MPa_to_GPa = 1e-3
sigma = [s*MPa_to_GPa for s in my_data["sigma"]]
```

For the sake of array calculations that are to be needed, data in the format shown in Fig. 9, is easier to work with, so the data is converted to that form as follows:

```
ep_vec = np.concatenate(epsilon_p)

log_ep_dot_vec = np.concatenate(
    [np.full(len(epsilon_p[i]), log_ep_dot[i])
     for i in range(len(log_ep_dot))]
)

T_star_vec = np.concatenate(T_star)
sigma_vec = np.concatenate(sigma)
```

To estimate $\boldsymbol{\theta}_0$ for the Johnson-Cook model, the mean of the MCMC samples from the fit with a strong prior on A (again, for $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$) is used:

```
import pandas

summary = pandas.read_csv(
    os.path.join("summaries",
                 "jc_MIDAS_pymc3_summary_strong_prior_on_A_bTQ09_fA075.csv"),
    index_col = 0
)

theta_0 = (summary.loc["A", "mean"]*MPa_to_GPa,
           summary.loc["B", "mean"]*MPa_to_GPa,
           summary.loc["n", "mean"],
           summary.loc["C", "mean"],
           summary.loc["m", "mean"])
```

At this point, one can begin to construct the inputs that will be needed for constrained minimization with the function `linprog` from the `optimize` submodule of SciPy.[37] These inputs consist of (1) a vector whose elements are the coefficients of the elements of $\Delta\boldsymbol{\theta}'_{min}$ and $\Delta\boldsymbol{\theta}'_{max}$ in Eq. 14 and (2) an array/vector pair that together characterizes the inequalities in Eq. 11. To construct these inputs, first $\mathbf{g}_{\sigma_{mdl}}$ from Eq. 14 is evaluated for $\boldsymbol{\theta}_0$ and the strains, strain rates, and temperatures from `my_data`:

```
g_sigma_mdl = jc_grad(ep_vec, log_ep_dot_vec, T_star_vec, *theta_0)
```

The columns of the array `g_sigma_mdl` are gradient vectors, each evaluated at a

given strain, strain rate, and temperature. To find the aforementioned coefficients of the elements of $\Delta\boldsymbol{\theta}'_{min}$ and $\Delta\boldsymbol{\theta}'_{max}$, one needs the sum of the elementwise absolute values of these vectors, which can be done as shown:

```
g_sigma_mdl_abs = np.fabs(g_sigma_mdl)
g_sigma_mdl_abs_sum = g_sigma_mdl_abs.sum(axis = 1)
```

The vector of coefficients, then is as follows:

```
coefficients = np.concatenate([g_sigma_mdl_abs_sum, g_sigma_mdl_abs_sum])
```

The first half of the vector `coefficients` is the coefficients for the elements of $\Delta\boldsymbol{\theta}'_{min}$, while the second half is the coefficients for the elements of $\Delta\boldsymbol{\theta}'_{max}$. In principle, since Eq. 14 is a function to be minimized, it can be multiplied by any nonzero prefactor without affecting the minimization. In practice, however, dividing it by the number of data points makes the numerical minimization more tractable. Accordingly,

```
num_data_pts = len(ep_vec)
coefficients /= num_data_pts
```

The inequalities in Eq. 14 need to be rearranged to fit the form needed by the function `linprog`, that is, $\mathbf{Au} \leq \mathbf{b}$. Here, $\mathbf{u}$ is a vector consisting of the elements of $\Delta\boldsymbol{\theta}_{min}$ followed by the elements of $\Delta\boldsymbol{\theta}_{max}$, and $\mathbf{A}$ is a matrix whose rows are coefficients of the elements of $\mathbf{u}$. The operator $\leq$ is here taken to operate elementwise, and $\mathbf{b}$ is another vector. To fit this format, Eq. 14 can be combined with Eqs. 12 and 13 and rearranged to obtain

$$\begin{aligned} &-\frac{1}{2}\left(\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}_j^{i_c}) + |\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}_j^{i_c})|\right)^T \Delta\boldsymbol{\theta}_{min} \\ &\quad + \frac{1}{2}\left(\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}_j^{i_c}) - |\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}_j^{i_c})|\right)^T \Delta\boldsymbol{\theta}_{max} \leq \sigma_j^{i_c} - \sigma_{mdl}(\mathbf{e}_j^{i_c}, \boldsymbol{\theta}_0) \end{aligned} \tag{15}$$

$$\begin{aligned} &\frac{1}{2}\left(\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}_j^{i_c}) - |\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}_j^{i_c})|\right)^T \Delta\boldsymbol{\theta}_{min} \\ &\quad - \frac{1}{2}\left(\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}_j^{i_c}) + |\mathbf{g}_{\sigma_{mdl}}(\mathbf{e}_j^{i_c})|\right)^T \Delta\boldsymbol{\theta}_{max} \leq -\left(\sigma_j^{i_c} - \sigma_{mdl}(\mathbf{e}_j^{i_c}, \boldsymbol{\theta}_0)\right) \end{aligned} \tag{16}$$

Accordingly, then, the matrix $\mathbf{A}$ and vector $\mathbf{b}$ can be constructed in Python as follows:

```
A_mat = np.empty((2*num_data_pts, len(coefficients)))
b_vec = np.empty(2*num_data_pts)
```

```
g_gabs_half_sum = 0.5*(g_sigma_mdl + g_sigma_mdl_abs)
g_gabs_half_diff = 0.5*(g_sigma_mdl - g_sigma_mdl_abs)
half_num_coeffs = int(len(coefficients)/2)
sigma_minus_sigma_mdl = sigma_vec - jc(ep_vec,
                                       log_ep_dot_vec,
                                       T_star_vec,
                                       *theta_0)

A_mat[:num_data_pts, :half_num_coeffs] = -g_gabs_half_sum.T
A_mat[:num_data_pts, half_num_coeffs:] = g_gabs_half_diff.T
b_vec[:num_data_pts] = sigma_minus_sigma_mdl

A_mat[num_data_pts:, :half_num_coeffs] = g_gabs_half_diff.T
A_mat[num_data_pts:, half_num_coeffs:] = -g_gabs_half_sum.T
b_vec[num_data_pts:] = -sigma_minus_sigma_mdl
```

At this point, the minimization can proceed. The default method used by `linprog` for minimization does not work for this problem, so a more robust alternative method, interior point, is used instead:

```
import scipy.optimize as so

result = so.linprog(coefficients, A_ub = A_mat, b_ub = b_vec,
                    method = "interior-point")

print("result.success = {}".format(result.success))
print("result.message = {}".format(result.message))

Delta_theta_min = result.x[:half_num_coeffs]
Delta_theta_max = result.x[half_num_coeffs:]

JC_param_lb = theta_0 - Delta_theta_min
JC_param_ub = theta_0 + Delta_theta_max

print("Est. spread for A: [{}, {}] (MPa)".format(JC_param_lb[0]/MPa_to_GPa,
                                                 JC_param_ub[0]/MPa_to_GPa))
print("Est. spread for B: [{}, {}] (MPa)".format(JC_param_lb[1]/MPa_to_GPa,
                                                 JC_param_ub[1]/MPa_to_GPa))
print("Est. spread for n: [{}, {}]".format(JC_param_lb[2], JC_param_ub[2]))
print("Est. spread for C: [{}, {}]".format(JC_param_lb[3], JC_param_ub[3]))
print("Est. spread for m: [{}, {}]".format(JC_param_lb[4], JC_param_ub[4]))
```

The output of the minimization is as follows:

```
result.success = True
result.message = Optimization terminated successfully.
Est. spread for A: [700.6180075481943, 700.6180075795291] (MPa)
Est. spread for B: [865.5153150776541, 865.5153153566544] (MPa)
Est. spread for n: [0.07201125850877127, 0.12337719049331143]
Est. spread for C: [0.004540558366680981, 0.007103479507105404]
Est. spread for m: [0.8742604532895236, 1.0512302974618548]
```

The resulting upper and lower bounds for the Johnson-Cook parameters (stored in the vectors `JC_param_lb` and `JC_param_ub`, respectively) have been estimated using a Taylor approximation. To see if these bounds are reasonable, the set $\boldsymbol{\Theta}$ will be taken to be the hyperrectangle with the corners `JC_param_lb` and `JC_param_ub`, and σ_{min} and σ_{max} will be estimated using Eqs. 8 and 9 (rather than the approximations in Eqs. 12 and 13). One can then determine how much of the flow stress data is actually bounded by σ_{min} and σ_{max}.

To do this, one first needs to create wrappers around the `jc` function that will work as objective functions for the `minimize` function from the `optimize` submodule of SciPy[37]:

```
def jc_for_min(ABnCm, ep, l_epdot, T_s):
    return jc(ep, l_epdot, T_s,
              ABnCm[0], ABnCm[1], ABnCm[2], ABnCm[3], ABnCm[4])

def jc_for_max(ABnCm, ep, l_epdot, T_s):
    return -jc_for_min(ABnCm, ep, l_epdot, T_s)
```

Since a *minimization* routine is used to find σ_{max}, `jc_for_max` is the negative of the Johnson-Cook flow stress. (Maximizing an objective function is the same as minimizing the negative of that function.)

One can then use the following `for` loop to generate estimates of σ_{min} and σ_{max} for each set of strain, strain rate, and temperature inputs, check how much of the data is within bounds, and then save the bounds to a Python pickle file (which is used to generate the plots in Ramsey[4] that show how much of the data are within bounds):

```
num_data_pts_in_bounds = 0

sigma_min = []
sigma_max = []

for i in range(len(epsilon_p_dot)):
    sigma_min.append(np.empty(len(epsilon_p[i])))
    sigma_max.append(np.empty(len(epsilon_p[i])))

    for j in range(len(epsilon_p[i])):
        result_min = so.minimize(jc_for_min,
                                 theta_0,
                                 args = (epsilon_p[i][j],
                                         log_ep_dot[i],
                                         T_star[i][j]),
                                 bounds = so.Bounds(JC_param_lb,
                                                    JC_param_ub))
```

```
        if not(result_min.success):
            print("Cannot find sigma_min for data point ({},{})!".format(i,j))

        result_max = so.minimize(jc_for_max,
                                 theta_0,
                                 args = (epsilon_p[i][j],
                                         log_ep_dot[i],
                                         T_star[i][j]),
                                 bounds = so.Bounds(JC_param_lb,
                                                    JC_param_ub))

        if not(result_max.success):
            print("Cannot find sigma_max for data point ({},{})!".format(i,j))

        sigma_min[i][j] = result_min.fun
        sigma_max[i][j] = -result_max.fun

        num_data_pts_in_bounds += \
           int(sigma_min[i][j] <= sigma[i][j] <= sigma_max[i][j])

print("Fraction of data points in bounds = {}".format(num_data_pts_in_bounds/
                                                      num_data_pts))
bssu.save_to_pickle_file(
    {"sigma_min": [s/MPa_to_GPa for s in sigma_min],
     "sigma_max": [s/MPa_to_GPa for s in sigma_max]},
    os.path.join(
        "pkl_data_files",
        "jc_MIDAS_pymc3_IPM_sigma_bounds_strong_prior_on_A_beta0.9_farea0.75.pkl.
            gz"
    )
)
```

The resulting text output is as follows:

```
Fraction of data points in bounds = 0.9994553376906318
```

Almost 100% of the data points are within the bounds.

Similar constrained optimizations to estimate bounds of parameters in the Johnson-Cook have been done for the rest of the combinations of β_{TQ} and f_{area} in Table 1, all for the case with a strong prior on A. Bounds have also been estimated for the parameters of the Zerilli-Armstrong (BCC) model, for the case where only the MIDAS data used to fit the Johnson-Cook model are included. Results for these cases are in Ramsey.[4]

7. Postprocessing of Model Fits

7.1 Plotting Priors with Posteriors

As a sanity check, one may compare the priors for the model parameters to their corresponding posteriors. If a posterior largely resembles its corresponding prior, this suggests that the posterior has been largely determined by the prior rather than the likelihood, which is a problem if a prior is only weakly informative and little more than an educated guess.

First, one needs to read in the samples of the posterior from an MCMC run, such as samples in the CSV file from the MCMC run of the Johnson-Cook model with $\beta_{TQ} = 0.9$, $f_{area} = 0.75$ and weakly informative priors. Here, the samples from such runs using PyStan and PyMC3 are read in to the Pandas data frames `jc_samples_pystan` and `jc_samples_pymc3`, respectively:

```
import pandas
import os

jc_samples_pystan = pandas.read_csv(
    os.path.join("samples",
                 "jc_MIDAS_pystan_samples_weak_prior_bTQ09_fA075.csv.gz"))

jc_samples_pymc3 = pandas.read_csv(
    os.path.join("samples",
                 "jc_MIDAS_pymc3_samples_weak_prior_bTQ09_fA075.csv.gz"))
```

The CSV files named have column headers corresponding to the names of model parameters (`"A"`, `"B"`, etc.), so the values in the columns of these CSV files, which contain the MCMC samples for those parameters, can be accessed as `jc_samples_pystan["A"]`, `jc_samples_pymc3["B"]`, and so on. However, the sequence of samples for model parameter $SD_{\sigma,1}$ (or `sd_sigma[1]` in the Stan specification file) is either `jc_samples_pystan["sd_sigma[0]"]` or `jc_samples_pymc3["sd_sigma__0"]`, with 0-based indexing being used.

The Python code for calculating the prior PDFs is as follows:

```
import numpy as np
import bayes_stress_strain_utils as bssu
from scipy.stats import norm, beta

JC_priors = bssu.read_from_json_file(
    os.path.join(os.pardir, "Other_data", "JC_priors.json")
)
```

```
prior_curves = {}
for pystan_param in jc_samples_pystan.columns:
    if pystan_param == "n":
        prior_x = np.linspace(0.0, 1.0, 100)
        prior_curves[pystan_param] = {
            "x": prior_x,
            "y": beta.pdf(prior_x,
                          JC_priors["n_alpha"], JC_priors["n_beta"])
        }
    elif pystan_param != "lp__":
        if pystan_param == "sd_sigma[0]":
            guess_mean = JC_priors["sd_sigma_guess_mean"][0]
            guess_sd = JC_priors["sd_sigma_guess_sd"][0]
            pymc3_param = "sd_sigma__0"
        elif pystan_param == "sd_sigma[1]":
            guess_mean = JC_priors["sd_sigma_guess_mean"][1]
            guess_sd = JC_priors["sd_sigma_guess_sd"][1]
            pymc3_param = "sd_sigma__1"
        else:
            guess_mean = JC_priors["{}_guess_mean".format(pystan_param)]
            guess_sd = JC_priors["{}_guess_sd".format(pystan_param)]
            pymc3_param = pystan_param

        prior_x_min = min([guess_mean - 3*guess_sd,
                           min(jc_samples_pystan[pystan_param]),
                           min(jc_samples_pymc3[pymc3_param])])

        prior_x_max = max([guess_mean + 3*guess_sd,
                           max(jc_samples_pystan[pystan_param]),
                           max(jc_samples_pymc3[pymc3_param])])

        prior_x = np.linspace(prior_x_min, prior_x_max, 100)

        prior_curves[pystan_param] = {
            "x": prior_x,
            "y": norm.pdf(prior_x, guess_mean, guess_sd)
        }
```

The `pdf` method of `beta` calculates the probability density for a beta distribution. Its first argument is an array of values for which the probability density is calculated, and the next two arguments are the α and β parameters of the distribution. The `pdf` method of `norm` is similar, except it calculates the probability density for a normal distribution, and its second and third arguments are the mean and standard deviation of the distribution. The dictionary `prior_curves` is used to store the x- and y-coordinates of points along the probability density curve for each parameter.

The following code plots the histograms for the marginal posterior PDFs of the Johnson-Cook model parameters and nuisance parameters $SD_{\sigma,1}$ and $SD_{\sigma,2}$, along

with their corresponding priors, after setting up the labels for the x-axes. In this code, the argument `density = True` is passed to the `hist` function so that the counts in the histogram bins are normalized such that the area under the histogram is 1. (A normalized histogram is more readily compared with a PDF, since the area under the whole PDF curve is also 1.) The argument `bins = "auto"` is passed to `hist` so that the number of bins in the histogram is determined from the samples, using an algorithm described by the NumPy developers.[44] Also, since the histograms of samples from PyStan and PyMC3 are to be overlapped, they are plotted not as bar charts, but rather with lines that show the outlines of the bars, by passing the argument `histtype = "step"` to the function `hist`. The resulting plots are shown in Fig. 12.

```
import matplotlib.pyplot as plt

# Setting default values for x-axis labels
x_labels = {pystan_param: pystan_param for
            pystan_param in jc_samples_pystan.columns}

# Modifying x-axis labels
for param in ["A", "B"]:
    x_labels[param] += " (MPa)"

x_labels["sd_sigma[0]"] = "$SD_{\\sigma,1}$ (MPa)"
x_labels["sd_sigma[1]"] = "$SD_{\\sigma,2}$ (MPa)"

# Plotting histograms with their associated priors
for pystan_param in jc_samples_pystan.columns:

    if pystan_param != "lp__":
        prior_x = prior_curves[pystan_param]["x"]
        prior_y = prior_curves[pystan_param]["y"]

        if pystan_param == "sd_sigma[0]":
            pymc3_param = "sd_sigma__0"
        elif pystan_param == "sd_sigma[1]":
            pymc3_param = "sd_sigma__1"
        else:
            pymc3_param = pystan_param

        plt.figure(figsize = (3,3.5))

        plt.hist(jc_samples_pystan[pystan_param],
                 density = True,
                 histtype = "step", linestyle = "dashed",
                 bins = "auto",
                 label = "PyStan")

        plt.hist(jc_samples_pymc3[pymc3_param],
                 density = True,
```

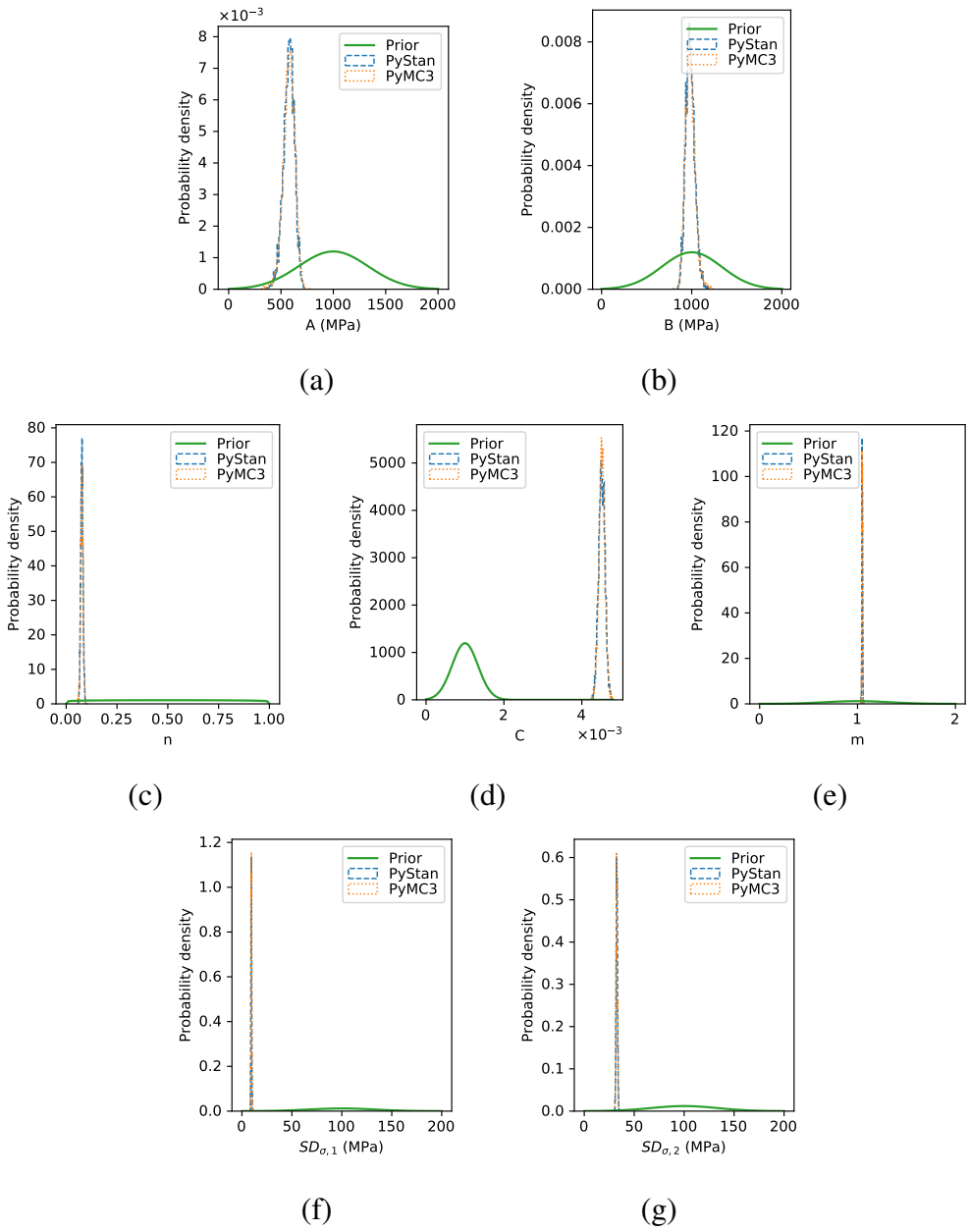


Fig. 12 Histograms approximating the posterior marginal PDFs of Johnson-Cook model parameters and nuisance parameters $SD_{\sigma,1}$ and $SD_{\sigma,2}$. These are generated from samples of PyStan and PyMC3 runs with $\beta_{TQ} = 0.9$, $f_{area} = 0.75$, and weakly informative priors. Priors are superimposed over the histograms.

```
                histtype = "step", linestyle = "dotted",
                bins = "auto",
                label = "PyMC3")

        plt.plot(prior_x, prior_y, linestyle = "solid",
                label = "Prior")

        plt.xlabel(x_labels[pystan_param])
        plt.ylabel("Probability density")

        # Use scientific notation for numbers outside the range
        # [1e-3, 1e4]
        plt.ticklabel_format(axis='both', style='sci',
                             scilimits=(-3,4), useMathText = True)

        plt.legend(loc = "best", labelspacing = 0.1)
        plt.tight_layout()

        out_pdf_name = \
            "jc_prior_vs_marg_posterior_for_{}_weak_prior_bTQ09_fA075.pdf".format(
            pymc3_param)

        plt.savefig(os.path.join("plot_files", out_pdf_name))

plt.close("all")
```

Further histograms of the marginal posteriors of the Johnson-Cook model parameters and nuisance parameters $SD_{\sigma,1}$ and $SD_{\sigma,2}$, as well their corresponding priors, for the case of a strongly informative prior on A, are shown in Ramsey.[4] Similar histograms are also presented in Ramsey[4] for the parameters of the Zerilli-Armstrong (BCC) model (and associated nuisance parameters), for both the fit to all available MIDAS data and the fit to the data used to fit the Johnson-Cook model.

7.2 Plotting Posteriors for Different Values of β_{TQ} and f_{area}

To crudely attempt to quantify the uncertainty due to variations in β_{TQ} and f_{area}, the marginal posterior PDFs determined for the values of β_{TQ} and f_{area} in Table 1 are compared. First, MCMC samples associated with these values are read in as follows. These samples are generated by PyStan and are for Johnson-Cook model fits with weakly informative priors:

```
import pandas
import os.path

bTQ_fA_strs = [
    ("0.9", "0.75"), ("0.9", "0.55"), ("0.9", "0.95"),
    ("0.6", "0.55"), ("0.6", "0.95")
]
```

```
jc_samples = {}

for bTQ, fA in bTQ_fA_strs:
    # Removing the decimal points from bTQ and fA, i.e.
    # 0.9 and 0.75 become 09 and 075
    bTQ_no_decimal = bTQ.replace(".", "")
    fA_no_decimal = fA.replace(".", "")

    csv_file_name = \
        "jc_MIDAS_pystan_samples_weak_prior_bTQ{}_fA{}.csv.gz".format(
        bTQ_no_decimal, fA_no_decimal)

    jc_samples[(bTQ,fA)] = pandas.read_csv(os.path.join(
        "samples", csv_file_name))
```

Here, `jc_samples` is a dictionary of Pandas data frames, where each frame is a set of MCMC samples associated with a pair of β_{TQ} and f_{area} values. Next, histograms for each pair of β_{TQ} and f_{area} values are computed from the MCMC samples:

```
import matplotlib.pyplot as plt

params = jc_samples[bTQ_fA_strs[0]].columns

# Setting up x-axis labels
x_labels = {param: param for param in params}

for param in ["A", "B"]:
    x_labels[param] += " (MPa)"

for i in range(2):
    x_labels["sd_sigma[{}]".format(i)] = \
        "$SD_{{\\sigma,{}}}$ (MPa)".format(i)

# Setting up legend labels
legend_labels = {}
for bTQ, fA in bTQ_fA_strs:
    legend_labels[(bTQ,fA)] = \
        "$\\beta_{{TQ}}$ = {}, $f_{{area}} = {}$".format(bTQ,fA)

linestyles = ["solid", "dashed", "dashdot", "dotted"]

# Plotting the actual superimposed histograms
for param in params:
    if param != "lp__":
        plt.figure(figsize = (3,3.5))

        plt.hist(jc_samples[bTQ_fA_strs[0]][param],
                 density = True,
                 histtype = "bar",
                 bins = "auto",
```

```
                label = legend_labels[bTQ_fA_strs[0]])

        # Adding histograms to plot
        for i, bTQ_fA in enumerate(bTQ_fA_strs[1:]):
            plt.hist(jc_samples[bTQ_fA][param],
                     density = True,
                     histtype = "step",
                     bins = "auto",
                     linestyle = linestyles[i],
                     label = legend_labels[bTQ_fA])

        plt.xlabel(x_labels[param])
        plt.ylabel("Probability density")

        # Use scientific notation for numbers outside the range
        # [1e-3, 1e4]
        plt.ticklabel_format(axis='both', style='sci',
                             scilimits=(-3,4), useMathText = True)

        ymin, ymax = plt.ylim()
        ymax += 0.5*(ymax - ymin)
        plt.ylim(ymax = ymax)

        plt.legend(loc = "best", labelspacing = 0.05, fontsize = 9)
        plt.tight_layout()

        # This turns "sd_sigma[0]" to "sd_sigma__0"
        file_name_param = param.replace("[", "__").replace("]", "")

        out_pdf_name = "jc_hists_for_{}_weak_prior_pystan.pdf".format(
                               file_name_param)

        plt.savefig(os.path.join("plot_files", out_pdf_name))

plt.close("all")
```

Again, the argument `density = True` is passed to the `hist` function so that the counts in the histogram bins are normalized such that the area under the histogram is 1, and the argument `bins = "auto"` is passed to `hist` so that the number of bins in the histogram is determined from the samples. Since these histograms are to be overlapped, all but the ones for $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ are plotted not as bar charts, but rather with lines that show the outlines of the bars, by passing the argument `histtype = "step"` to the function `hist`. The resulting plots are shown in Fig. 13.

Further histograms of the marginal posteriors of the Johnson-Cook model parameters and nuisance parameters $SD_{\sigma,1}$ and $SD_{\sigma,2}$ for the case of a strongly informative prior on A, are shown in Ramsey.[4] Similar histograms are also presented in Ramsey[4]

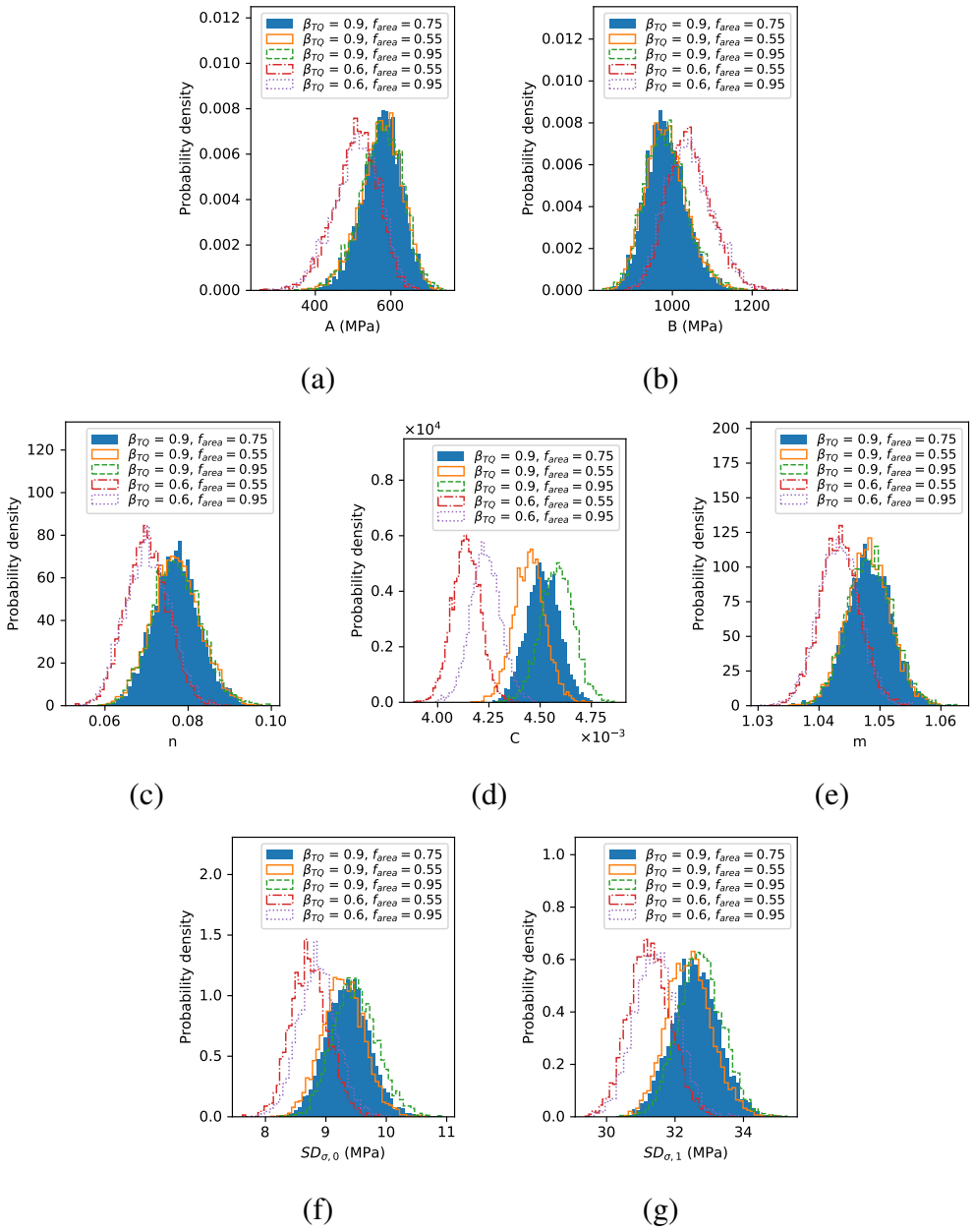


Fig. 13 Histograms approximating the posterior marginal PDFs of Johnson-Cook model parameters and nuisance parameters $SD_{\sigma,1}$ and $SD_{\sigma,2}$. These are generated from samples of PyStan MCMC runs with the values of β_{TQ} and f_{area} in Table 1, and weakly informative priors.

for the parameters of the Zerilli-Armstrong (BCC) model (and associated nuisance parameters), for both the fit to all available MIDAS data and the fit to the data used to fit the Johnson-Cook model.

7.3 Plotting PPDs and PFPs with Experimental Data

To generate PPDs and PFPs for the Johnson-Cook model, one needs the samples from an MCMC fit of the model. Accordingly, samples from the Johnson-Cook fits with weakly-informative priors are to be loaded, as they have been in Section 7.2, into a dictionary of data frames named `jc_samples`, where each frame is associated with a tuple such as `("0.9", "0.75")`, which represents a pair of β_{TQ} and f_{area} values. One also needs the data and model used with the MCMC run that generated the samples. The data used to fit the Johnson-Cook model have been stored in pickle files (as discussed in Section 6.2), and these are loaded again, with `bTQ_fA_strs` defined as in Section 7.2. The stress-strain data in `Main_data_for_JC_pymc3.pkl.gz` is in a more convenient format than that in `Main_data_for_JC_pystan.pkl.gz`, since the strains for each stress-strain curve are in their own separate arrays rather than concatenated together in one long array, so it is used even with the MCMC samples from PyStan:

```
import bayes_stress_strain_utils as bssu

main_data = bssu.read_from_pickle_file(
    os.path.join("pkl_data_files",
                 "Main_data_for_JC_pymc3.pkl.gz"))

temp_data = {}
for bTQ, fA in bTQ_fA_strs:
    temp_data[(bTQ, fA)] = bssu.read_from_pickle_file(
        os.path.join("pkl_data_files",
                     "T_beta{}_farea{}_JC.pkl.gz".format(bTQ,
                                                         fA)))
```

So that the procedure for generating PPDs and PFPs is nearly identical for both the Johnson-Cook and Zerilli-Armstrong models, a wrapper function named `sigma_model_func` is used, rather than directly using the `jc` and `za_bcc` functions (from the modules `jc` and `za_bcc`, respectively). For the Johnson-Cook model, this wrapper function is the same as the one used in Section 5.2. Since the MCMC samples differ for different values of β_{TQ} and f_{area}, different values of the `theta_model` used in `sigma_model_func` are needed for those values, so a dictionary of the needed values is created. A dictionary of samples of $SD_{\sigma,1}$ and

$SD_{\sigma,2}$ is generated as well for those values of β_{TQ} and f_{area}:

```
theta_model_dict = {}
sd_sigma_dict = {}

for bTQ_fA in bTQ_fA_strs:
    curr_samples = jc_samples[bTQ_fA]

    theta_model_dict[bTQ_fA] = {
        "A": curr_samples["A"],
        "B": curr_samples["B"],
        "n": curr_samples["n"],
        "C": curr_samples["C"],
        "m": curr_samples["m"],
        "epsilon_p_dot_0": main_data["epsilon_p_dot_0"],
        "T_room": main_data["T_room"],
        "T_melt": main_data["T_melt"]
    }

    sd_sigma_dict[bTQ_fA] = [
        curr_samples["sd_sigma[0]"],
        curr_samples["sd_sigma[1]"]
    ]
```

At this point, one may proceed to generate samples of PPDs and PFPs with a loop that partly resembles the main loop in the `model` block of Stan specification file for the Johnson-Cook model, `jc.stan`.[10] However, rather than store the samples from all the PPDs and PFPs, which could use a significant amount of memory since there are 4000 samples for each of the roughly 2000 data points that make up the stress-strain curve data, select statistics from the PPDs and PFPs are kept instead. For each PPD, these statistics are the mean and the bounds of the 95% highest density interval (HDI), which is the interval such that 1) the probability that a value is in this interval is 95% and 2) the values within this interval all have higher probability densities than values outside of it.[12] These statistics are computed and stored in the dictionaries of arrays `ppd_mean`, `ppd_hdi_min`, and `ppd_hdi_max`, which are defined in the following Python code. For each PFP, only the bounds of the 95% HDI are computed. These bounds are in the dictionaries of arrays `pfp_hdi_min` and `pfp_hdi_max`, which are also defined in the following Python code:

```
ppd_mean = {}
ppd_hdi_min = {}
ppd_hdi_max = {}

pfp_hdi_min = {}
pfp_hdi_max = {}
```

```
epsilon_p = main_data["epsilon_p"]
epsilon_p_dot = main_data["epsilon_p_dot"]
num_curves = len(epsilon_p_dot)

for bTQ_fA in bTQ_fA_strs:

    T = temp_data[bTQ_fA]
    theta_model = theta_model_dict[bTQ_fA]
    sd_sigma = sd_sigma_dict[bTQ_fA]

    ppd_mean[bTQ_fA] = []
    ppd_hdi_min[bTQ_fA] = []
    ppd_hdi_max[bTQ_fA] = []

    pfp_hdi_min[bTQ_fA] = []
    pfp_hdi_max[bTQ_fA] = []

    for curve_ind in range(num_curves):

        if epsilon_p_dot[curve_ind] <= 1.0:
            curr_sd_sigma = sd_sigma[0]
        else:
            curr_sd_sigma = sd_sigma[1]

        pts_per_curve = len(epsilon_p[curve_ind])

        curr_ppd_mean = np.empty(pts_per_curve)
        curr_ppd_hdi_min = np.empty(pts_per_curve)
        curr_ppd_hdi_max = np.empty(pts_per_curve)

        curr_pfp_hdi_min = np.empty(pts_per_curve)
        curr_pfp_hdi_max = np.empty(pts_per_curve)

        for i in range(pts_per_curve):
            curr_pfp = \
                sigma_model_func(epsilon_p[curve_ind][i],
                                 epsilon_p_dot[curve_ind],
                                 T[curve_ind][i],
                                 theta_model)

            curr_ppd = curr_pfp + \
                np.random.randn(len(curr_pfp))*curr_sd_sigma

            curr_ppd_mean[i] = np.mean(curr_ppd)

            # By default, the hdi function computes the bounds
            # of the 95% HDI.
            curr_ppd_hdi_range = bssu.hdi(curr_ppd)
            curr_ppd_hdi_min[i] = curr_ppd_hdi_range[0]
            curr_ppd_hdi_max[i] = curr_ppd_hdi_range[1]

            curr_pfp_hdi_range = bssu.hdi(curr_pfp)
            curr_pfp_hdi_min[i] = curr_pfp_hdi_range[0]
            curr_pfp_hdi_max[i] = curr_pfp_hdi_range[1]
```

```
        ppd_mean[bTQ_fA].append(curr_ppd_mean)
        ppd_hdi_min[bTQ_fA].append(curr_ppd_hdi_min)
        ppd_hdi_max[bTQ_fA].append(curr_ppd_hdi_max)

        pfp_hdi_min[bTQ_fA].append(curr_pfp_hdi_min)
        pfp_hdi_max[bTQ_fA].append(curr_pfp_hdi_max)
```

The variable `curr_ppd` is a 1-D array of samples of the PPD associated with the strain and temperature values `epsilon_p[curve_ind][i]` and `T[curve_ind][i]`. Each element of `curr_ppd` corresponds to a value of $\sigma_j^{i_c,pred}(\epsilon_j^{i_c}, \dot{\epsilon}_p^{i_c}, T_j^{i_c})$ from Eq. 6, where i_c and j are *fixed* for *all* of the elements of `curr_ppd`. Element `curr_ppd[q]` is determined from the q^{th} MCMC sample of the model parameters. Each sample `curr_ppd[q]` of the PPD is drawn from a normal distribution, hence the presence of the function `randn`, which is used in the same way as it is used in Sections 5.2 and 5.5. Element `curr_pfp[q]` is, of course, a sample of the PFP determined from the q^{th} MCMC sample of the model parameters. Given the previous code, `ppd_mean[("0.9", "0.75")][i][j]` is an estimate of the mean of the PPD associated with the strain rate `epsilon_p_dot[i]` and the strain and temperature values `epsilon_p[i][j]` and `T[i][j]`, provided that $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$, while the expressions `ppd_hdi_min[("0.9", "0.75")][i][j]`, and `ppd_hdi_max[("0.9", "0.75")][i]` are estimates of the bounds of the 95% HDI of that PPD. These bounds are calculated via the function `hdi` from the `bayes_stress_strain_utils` module, the source of which is in Appendix C.

The code for plotting the means and 95% HDI bounds along with the experimental data is as follows:

```
import matplotlib.pyplot as plt

T_init_str = ["298", "298",  "298",  "298", "473",  "673",  "873"]
sigma = main_data["sigma"]

# Setting up line types and colors

line_types_mean_and_hdi = ["solid", "dashed", "dashdot", "dotted"]

# For beta_TQ = 0.9, f_area = 0.75
color_hdi_only = "skyblue"
color_mean_only = "purple"
```

```
# For other beta_TQ and f_area pairs
colors_mean_and_hdi = ["red", "brown", "green", "blue"]

# For experimental data
color_data = "black"

# Setting up legend labels
bTQ_fA_legend_strs = [
    "$\\beta_{{TQ}}$ = {}, $f_{{area}}$ = {}".format(bTQ,fA)
    for bTQ, fA in bTQ_fA_strs
]

legend_label_first_hdi = "95% HDI, {}".format(bTQ_fA_legend_strs[0])
legend_label_first_mean = "Mean, {}".format(bTQ_fA_legend_strs[0])

legend_labels_rest_of_bTQ_fA = [
    "Mean & 95% HDI, {}".format(bTQ_fA_lgd_str)
    for bTQ_fA_lgd_str in bTQ_fA_legend_strs[1:]
]

legend_label_data = "Exp. Data"

# No extra space needed for the legend for low strain rates, only
# for plots with high-strain-rate data
space_for_legend = [0.0]*2 + [0.5]*5

# Plotting HDI of PPDs
for curve_ind in range(num_curves):

    plt.figure(figsize = (4.375,4.25))

    curr_epsilon_p = epsilon_p[curve_ind]
    curr_sigma = sigma[curve_ind]

    curr_hdi_min = {}
    curr_hdi_max = {}
    for bTQ_fA in bTQ_fA_strs:
        curr_hdi_min[bTQ_fA] = ppd_hdi_min[bTQ_fA][curve_ind]
        curr_hdi_max[bTQ_fA] = ppd_hdi_max[bTQ_fA][curve_ind]

    # Plot 95% HDI for beta_TQ = 0.9, f_area = 0.75 as shaded region
    hdi_shade = plt.fill_between(curr_epsilon_p,
                                 curr_hdi_min[bTQ_fA_strs[0]],
                                 curr_hdi_max[bTQ_fA_strs[0]],
                                 color = color_hdi_only)

    # Plot mean for beta_TQ = 0.9, f_area = 0.75 as lines
    mean_line = plt.plot(curr_epsilon_p,
                         ppd_mean[bTQ_fA_strs[0]][curve_ind],
                         linestyle = "solid",
                         marker = "x", markersize = 5,
                         markevery = int(round(len(curr_epsilon_p)/25)),
                         color = color_mean_only)[0]
```

```
# Plot mean and bounds of 95% HDI as lines
hdi_lines = []
for i, bTQ_fA in enumerate(bTQ_fA_strs[1:]):
    hdi_line = plt.plot(curr_epsilon_p, curr_hdi_min[bTQ_fA],
                        linestyle = line_types_mean_and_hdi[i],
                        color = colors_mean_and_hdi[i])[0]

    hdi_lines.append(hdi_line)

    plt.plot(curr_epsilon_p, ppd_mean[bTQ_fA][curve_ind],
             linestyle = line_types_mean_and_hdi[i],
             color = colors_mean_and_hdi[i])

    plt.plot(curr_epsilon_p, curr_hdi_max[bTQ_fA],
             linestyle = line_types_mean_and_hdi[i],
             color = colors_mean_and_hdi[i])

# Plot experimental data
data_pts = plt.plot(curr_epsilon_p, curr_sigma,
                    linestyle = "None",
                    marker = ".", markersize = 2,
                    color = color_data)[0]

plt.title("{} K, {}/s".format(T_init_str[curve_ind],
                              epsilon_p_dot[curve_ind]))

plt.xlabel("$\\epsilon_p$")
plt.ylabel("$\\sigma$ (MPa)")

plt.legend([hdi_shade] + [mean_line] + hdi_lines + [data_pts],
           [legend_label_first_hdi] + \
           [legend_label_first_mean] + \
           legend_labels_rest_of_bTQ_fA + \
           [legend_label_data],
           loc = "lower right",
           labelspacing = 0.05,
           fontsize = 9)

ymin, ymax = plt.ylim()
ymin -= space_for_legend[curve_ind]*(ymax - ymin)
plt.ylim(ymin = ymin)

plt.tight_layout()

out_pdf_name = \
    "jc_hdi_edot{}_T{}_weak_prior_pystan.pdf".format(
        epsilon_p_dot[curve_ind],
        T_init_str[curve_ind])

plt.savefig(os.path.join("plot_files", out_pdf_name))
```

The resulting plots are shown in Figs. 14 and 15. Plots showing the 95% HDI bounds of the PFPs are shown in Figs. 16 and 17. The code to generate these plots

is very similar to the code that plots the statistics of the PPDs, so it is not shown.

Estimates for the mean and bounds of the 95% HDIs of the PPDs and PFPs have also been done for the Johnson-Cook model with a strong prior on A and for the Zerilli-Armstrong (BCC) model fitted to all available MIDAS data and the fit to the data used to fit the Johnson-Cook model. These are shown in Ramsey.[4]

7.4 Determining Correlation Matrices

In addition to statistics for the marginal PDFs of the model parameters, one may also need information on how the PDFs of these parameters are correlated, especially if one intends to use these PDFs as input to uncertainty propagation analyses. For example, when the software Dakota is used for such analyses, it takes as input either a correlation or rank correlation matrix, depending on the method of uncertainty propagation used.[45] Both of these are fairly simple to calculate for Pandas data frames. For the Johnson-Cook model fit by PyStan with weakly informative priors, $\beta_{TQ} = 0.9$, and $f_{area} = 0.75$, the correlation matrix may be evaluated as follows:

```
import pandas
import os.path

jc_samples = pandas.read_csv(
    os.path.join("samples",
                 "jc_MIDAS_pystan_samples_weak_prior_bTQ09_fA075.csv.gz"))

corr_mat_jc = jc_samples.corr()
print(corr_mat_jc)
```

The method `corr` calculates the Pearson correlation coefficient for each pair of columns in the data frame `jc_samples` and returns a square matrix where each element is the correlation coefficient for each column pair. Since each column in `jc_samples` (except for the column for `lp__`) is a sequence of MCMC samples for each model parameter, each entry in the matrix represents the correlation between the random distributions of a pair of parameters.

The calculation of the Spearman rank correlation matrix is similarly trivial:

```
rcorr_mat_jc = jc_samples.corr(method = "spearman")
print(rcorr_mat_jc)
```

Again, the function `corr` is used. However, when the argument "`method = "spearman"`" is used, for each element of a column in `jc_samples`, it as-

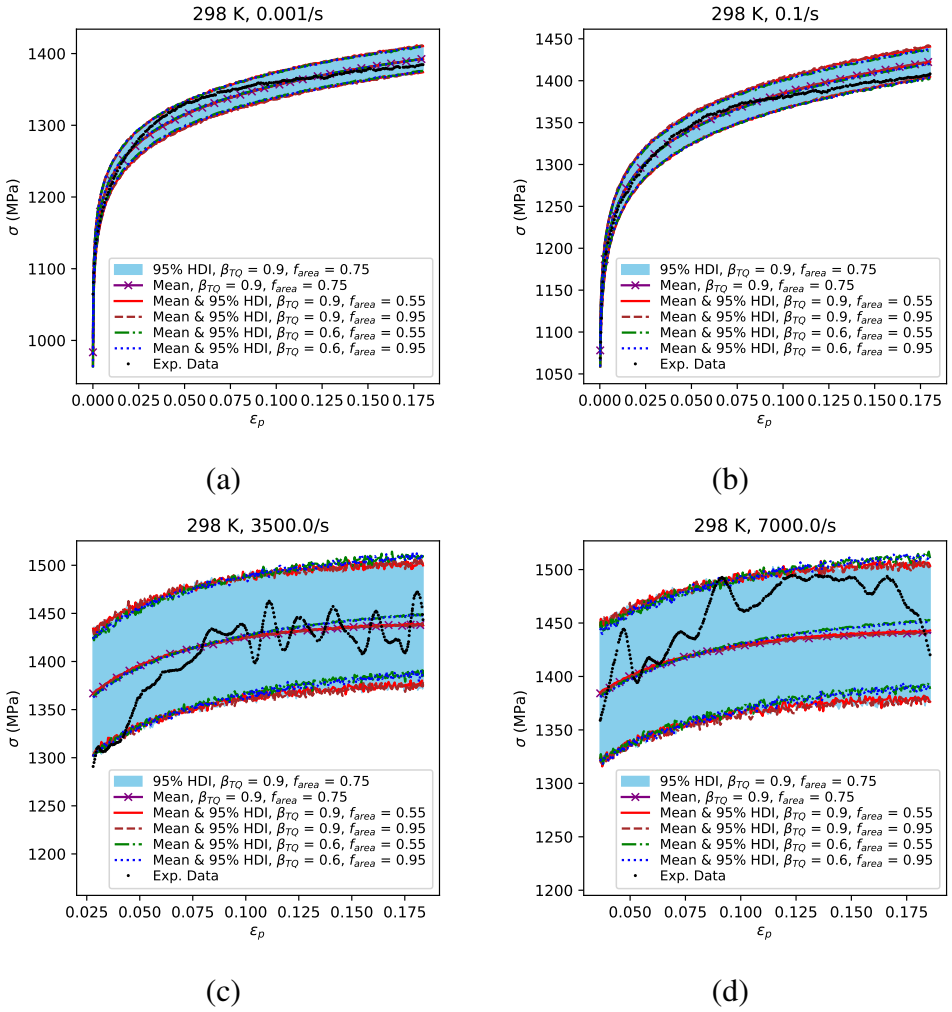


Fig. 14 Stress-strain data for initial sample temperatures of 298 K, along with estimates of the mean and the 95% HDI for PPDs generated from samples of PyStan MCMC runs for the Johnson-Cook model with weakly informative priors. The 95% HDI for $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ is plotted as a shaded region between the minimum and maximum of the HDI.

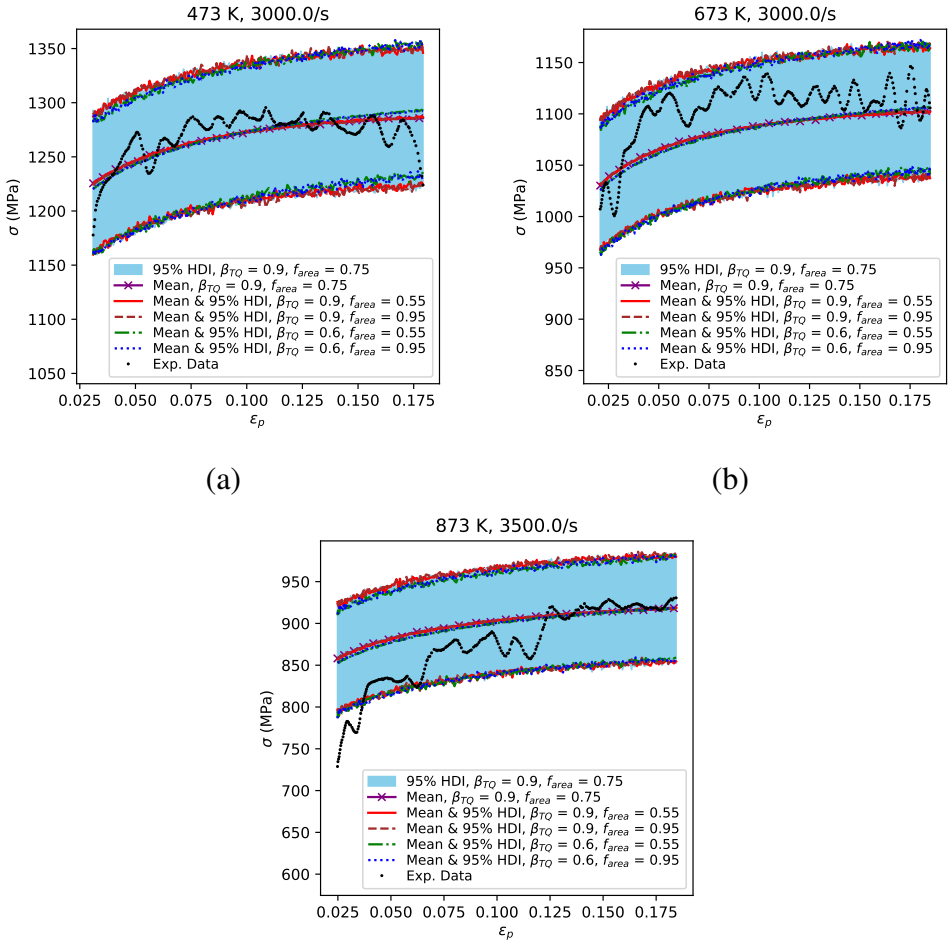


Fig. 15 Stress-strain data for high initial sample temperatures along with estimates of the mean and the 95% HDI for PPDs generated from samples of PyStan MCMC runs for the Johnson-Cook model with weakly informative priors. The 95% HDI for $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ is plotted as a shaded region between the minimum and maximum of the HDI.

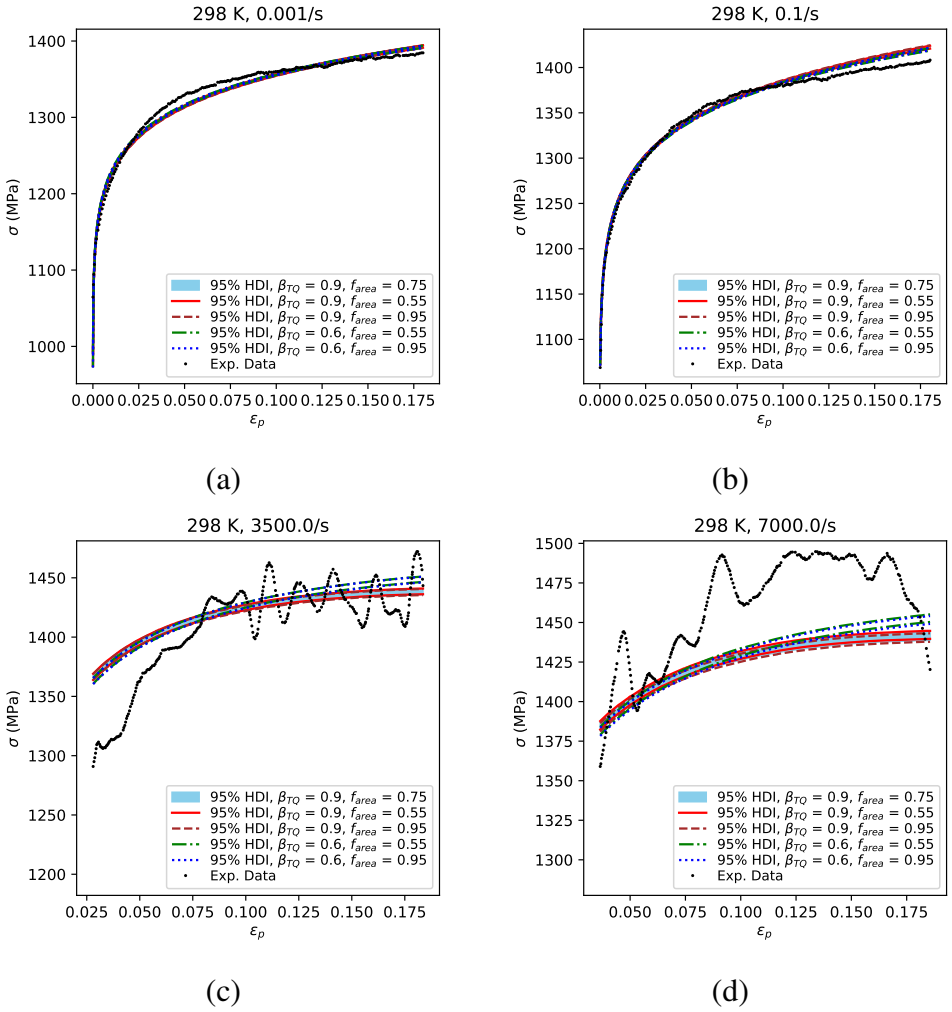


Fig. 16 Stress-strain data for initial sample temperatures of 298 K, along with estimates of the 95% HDI for PFPs of model predictions generated from samples of PyStan MCMC runs for the Johnson-Cook model with weakly informative priors. The 95% HDI for $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ is plotted as a shaded region between the minimum and maximum of the HDI.

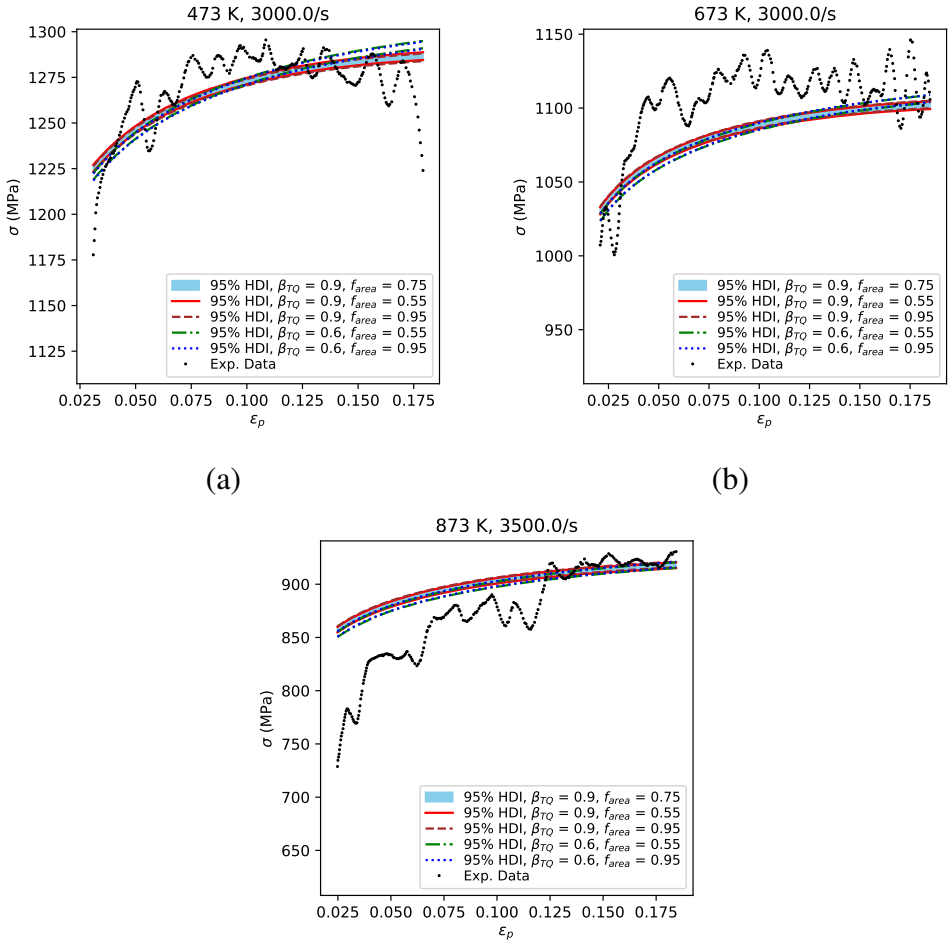


Fig. 17 Stress-strain data for high initial sample temperatures along with estimates of the 95% HDI for PFPs generated from samples of PyStan MCMC runs for the Johnson-Cook model with weakly informative priors. The 95% HDI for $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ is plotted as a shaded region between the minimum and maximum of the HDI.

signs a rank, such that the lowest rank, 1, is assigned to the smallest number in the column, the rank of 2 to the next smallest number in the column, and so on. Each column, then, is associated with a sequence of integer ranks. When the Spearman rank correlation coefficient is applied to a pair of columns, it replaces each column with its corresponding sequence of ranks, and then applies the Pearson correlation coefficient to the sequences of ranks.[46,47]

Both the correlation matrix `corr_mat_jc` and the rank correlation matrix `rcorr_mat_jc` may be saved to CSV files using their `to_csv` methods.

8. Conclusions

This report describes a workflow, based on PyStan, PyMC3, SciPy, and the Python scripting language, that has been used to obtain information on strength model parameters in RHA that can be used in uncertainty propagation analyses. This workflow covers several issues:

- testing Bayesian models, which can uncover potential problems such as the need to provide explicit initial values in some cases (e.g., the Zerilli-Armstrong [BCC] model);
- approximating the temperature rise in the samples being deformed in stress-strain experiments, noting how some of the assumptions in the approximations may affect the estimated marginal PDFs of the model parameters;
- keeping track of warning messages and other diagnostics from Bayesian software tools, noting what to do to address them;
- estimating bounds on model parameters via an approximate IPM approach;
- generating samples of a PPD and plotting statistics of them in a way that can be used to evaluate the fit of a strength model to experimental data; and
- accounting for *correlations* in the random distributions of model parameters, especially in a form that can be used as input for software tools that do uncertainty propagation, such as Dakota.[45]

It is hoped that this workflow may serve as a source of example code for other ARL researchers who wish to obtain results that facilitate uncertainty quantification.

9. References

1. Python Software Foundation. Welcome to python.org. c2018 [accessed 2018 May]. https://www.python.org.

2. PyStan developers. PyStan: The Python interface to Stan. c2018 [accessed 2018 May]. http://pystan.readthedocs.io.

3. PyMC Development Team. PyMC3 documentation. c2018 [accessed 2018 Mar]. http://docs.pymc.io/.

4. Ramsey JJ. Quantifying uncertainties in parameterizations of strength models of rolled homogeneous armor: part 1, overview. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2019 Sep. Report No.: ARL-TR-8826.

5. Benck RF. Quasi-static tensile stress strain curves: II, rolled homogeneous armor. Aberdeen Proving Ground (MD): Ballistic Research Laboratories (US); 1976 Nov. Report No.: 2703.

6. Rittel D, Zhang LH, Osovski S. The dependence of the Taylor-Quinney coefficient on the dynamic loading mode. Journal of the Mechanics and Physics of Solids. 2017;107:96–114.

7. Johnson GR, Cook WH. A constitutive model and data for metals subjected to large strains, high strain rates and high temperatures. In: Seventh international symposium on ballistics: Proceedings; 1983 Apr; The Hague (Netherlands). American Defense Preparedness Association; 1983. p. 541–547.

8. Zerilli FJ, Armstrong RW. Dislocation-mechanics-based constitutive relations for material dynamics calculations. Journal of Applied Physics. 1987;61(5):1816–1825.

9. Gray GT III, Chen SR, Wright W, Lopez MF. Constitutive equations for annealed metals under compression at high strain rates and high temperatures. Los Alamos (NM): Los Alamos National Laboratory; 1994 Jan. Report No.: LA-12669-MS.

10. Gelman A, Carlin JB, Stern HS, Dunson DB, Vehtari A, Rubin DB. Bayesian data analysis. 3rd ed. Boca Raton (FL): CRC Press; 2013.

11. Chowdhary K, Najm HN. Data free inference with processed data products. Statistics and Computing. 2016;26(1):149–169.

12. Kruschke JK. Doing Bayesian data analysis: a tutorial with R, JAGS, and Stan. 2nd ed. Waltham (MA): Academic Press; 2015.

13. Betancourt M. A conceptual introduction to Hamiltonian Monte Carlo. c2017 [accessed 2018 Mar]. https://arxiv.org/abs/1701.02434.

14. Hoffman MD, Gelman A. The no-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. Journal of Machine Learning Research. 2014;15(1).

15. Crespo LG, Kenny SP, Giesy DP. Calibration of predictor models using multiple validation experiments. In: 17th AIAA Non-Deterministic Approaches Conference; (AIAA SciTech Forum; no. AIAA 2015-0659) American Institute of Aeronautics and Astronautics; 2015.

16. Crespo LG, Kenny SP, Giesy DP. Interval predictor models with a linear parameter dependency. Journal of Verification, Validation and Uncertainty Quantification. 2016;1(2):021007.

17. PyStan developers. PyStan: detailed installation instructions. c2018 [accessed 2018 Mar]. http://pystan.readthedocs.io/en/latest/installation_beginner.html.

18. Salvatier J, Wiecki TV, Fonnesbeck C. Getting started with PyMC3. c2018 [accessed 2018 Mar]. http://docs.pymc.io/notebooks/getting_started.

19. Anaconda, Inc. Anaconda. c2018 [accessed 2018 Mar]. https://anaconda.com.

20. Goodrich B. R session aborted. c2019 May [accessed 2019 Sep]. https://discourse.mc-stan.org/t/r-session-aborted/6655/12.

21. Anaconda, Inc. Anaconda installation. c2018 [accessed 2018 Mar]. https://docs.anaconda.com/anaconda/install/.

22. Anaconda, Inc. Getting started with Navigator. c2018 [accessed 2018 Mar]. https://docs.anaconda.com/anaconda/navigator/getting-started.

23. oorc06. Can't pickle cvm objects #3068. c2018 [accessed 2018Aug]. https://github.com/pymc-devs/pymc3/issues/3068.

24. Project Jupyter. Project Jupyter. c2018 [accessed 2018 Mar]. http://jupyter.org/.

25. Spyder developers. The scientific Python development environment. c2018 [accessed 2018 Mar]. https://spyder-ide.github.io/.

26. Stan Development Team. Emacs support for Stan. c2017 [accessed 2018 Mar]. https://github.com/stan-dev/stan-mode.

27. Lerch M. mc-stan.vim. c2015 [accessed 2018 Mar]. https://github.com/mdlerch/mc-stan.vim.

28. Ramsey JJ. Quantifying uncertainties in parameterizations of strength models of rolled homogeneous armor: part 2, R-based workflow. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2019 Sep. Report No.: ARL-TR-8827.

29. Free Software Foundation. GNU gzip. c2018 [accessed 2018 May]. https://www.gnu.org/software/gzip/.

30. Ramsey JJ. split_potential_scale_reduction can yield spurious nan values #441. c2018 [accessed 2018 Mar]. https://github.com/stan-dev/pystan/issues/441.

31. Ramsey JJ. Issue #427: Integrate Betancourt's stan_utility functions into PyStan #433. c2018 [accessed 2018Mar]. https://github.com/stan-dev/pystan/pull/433.

32. PyStan developers. PyStan: the Python interface to Stan, API. c2018 [accessed 2018 Mar]. http://pystan.readthedocs.io/en/latest/api.html.

33. Betancourt M. Robust PyStan workflow. c2017 [accessed 2018 Mar]. http://mc-stan.org/users/documentation/case-studies.html#robust-pystan-workflow.

34. Stan Development Team. Brief guide to stan's warnings. c2018 [accessed 2018 May]. http://mc-stan.org/misc/warnings.html.

35. Hubert W. Meyer J, Kleponis DS. An analysis of parameters for the Johnson-Cook strength model for 2-in-thick rolled homogeneous armor. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2001 June. Report No.: ARL-TR-2528.

36. Pandas developers. pandas: powerful Python data analysis toolkit. c2017 [accessed 2018 May]. http://pandas.pydata.org/pandas-docs/stable/index.html.

37. SciPy developers. SciPy. c2018 [accessed 2018 May]. http://www.scipy.org/.

38. Stan Development Team. Stan modeling language user's guide and reference manual. 2017 Dec.

39. Hartikainen A. 1-based indexing #431. c2018 [accessed 2018 May]. https://github.com/stan-dev/pystan/pull/431.

40. Nelson A. np.savetxt raises an exception when passed a file handle that was opened with 'w' #6356. c2015 [accessed 2018 June]. https://github.com/numpy/numpy/issues/6356.

41. Brannigan L. Output samples to pandas dataframe #425. c2018 [accessed 2018 Mar]. https://github.com/stan-dev/pystan/pull/425.

42. Mathews JH, Fink KD. Numerical methods using Matlab. 4th ed. Upper Saddle River (NJ): Pearson Prentice Hall; 2004.

43. SymPy development team. SymPy. c2018 [accessed 2019 Mar]. http://www.sympy.org/.

44. NumPy developers. numpy.histogram: NumPy v1.14 manual. c2018 [accessed 2018 June]. https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html.

45. Adams BM et al. Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 6.8 reference manual. Albuquerque (NM): Sandia National Laboratories; 2018 May.

46. Pandas developers. pandas.dataframe.corr. c2017 [accessed 2018 June]. http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.corr.html.

47. Sprent P. Applied nonparametric statistical methods. New York (NY): Chapman and Hall; 1989.

Appendix A. Data Tables

These are tables of the data that have been used in Bayesian analyses of strength models of rolled homogeneous armor (RHA). Table A-1 contains values for the specific heat of body-centered cubic (BCC) iron—which is assumed to approximate the specific heat of RHA—as a function of temperature. In this table, the specific heat values are for constant volume, except for values for temperatures above 773 K, where only values for constant pressure are available. The specific heat values are converted from molar heat capacity values from Austin[1] using the molar mass of iron taken from the CRC Handbook,[2] 55.845 g/mol. Tables A-2 through A-10 contain the stress-strain data for RHA that comes from the Material Implementation, Database, and Analysis Source (MIDAS).[3] The original source for these data is Gray et al.,[4] who have obtained high-strain-rate data with a split Hopkinson pressure bar and low-strain-rate data (where the plastic strain rate is no greater than 1/s) with "either an Instron or an MTS testing system". However, the original published data are engineering stress and strain, while in the MIDAS database, it has been corrected to true stress and true plastic strain.[5]

Table A-1 Specific heat of BCC iron versus temperature

Temp. (K)	Spec. heat (J/kg · K)	Temp. (K)	Spec. heat (J/kg · K)	Temp. (K)	Spec. heat (J/kg · K)	Temp. (K)	Spec. heat (J/kg · K)	Temp. (K)	Spec. heat (J/kg · K)
20	4.123	200	382.356	323	454.329	573	565.287	1023	1154.566
30	11.246	225	400.349	333	457.328	623	583.281	1033	1341.245
40	27.515	250	419.092	343	459.577	673	602.773	1073	877.170
50	53.230	273.1	430.338	353	461.826	723	623.016	1123	812.694
75	134.949	283	436.336	363	464.825	773	647.756	1173	778.957
100	212.920	293	442.334	373	470.823	823	718.230		
125	272.148	298	444.583	423	494.814	873	790.203		
150	322.379	303	447.582	473	519.555	923	871.172		
175	356.866	313	451.330	523	541.296	973	962.638		

[1]Austin JB. Heat capacity of iron: a review. Industrial & Engineering Chemistry. 1932;24(11):1225–1235.

[2]Rumble J, editor. CRC handbook of chemistry and physics. 98th ed. Boca Raton (FL): CRC Press; 2017.

[3]Lawrence Livermore National Laboratory. MIDAS: Material implementation, database, and analysis source. c2018 [accessed 2018 Mar]. `https://pls.llnl.gov/people/divisions/physics-division/condensed-matter-science-section/eos-and-materials-theory-group/projects/midas-material-implementation-database-and-analysis-source`.

[4]Gray GT III, Chen SR, Wright W, Lopez MF. Constitutive equations for annealed metals under compression at high strain rates and high temperatures. Los Alamos (NM): Los Alamos National Laboratory; 1994 Jan. Report No.: LA-12669-MS.

[5]Florando J. Lawrence Livermore National Laboratory, Livermore, CA. Personal communication, 2017.

Table A-2 Flow stress versus plastic strain of RHA for initial temperature 77 K and plastic strain rate 0.001/s

Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)
0.000062	1552.7	0.032648	1764.6	0.067995	1854.4	0.104813	1930	0.143498	1986.7
0.00022	1566.5	0.033594	1765	0.068757	1855.3	0.105444	1931.3	0.144944	1987.5
0.000535	1582.5	0.034303	1768	0.069571	1857.4	0.107179	1930.5	0.145548	1989.3
0.000903	1597.1	0.035066	1770.6	0.070412	1857.9	0.108177	1933.9	0.146337	1989.3
0.001376	1609.2	0.035828	1771.9	0.071385	1859.2	0.109045	1934.4	0.147125	1991
0.001875	1621.7	0.036563	1774.1	0.072199	1862.2	0.109912	1938.7	0.147835	1991.4
0.002453	1631.6	0.037483	1776.2	0.072856	1864.3	0.110779	1940.8	0.149359	1994.5
0.003294	1642.4	0.039034	1779.7	0.07375	1865.2	0.111699	1943	0.150568	1995.8
0.003872	1650.2	0.03977	1781	0.074486	1867.4	0.112881	1944.3	0.151093	1995.8
0.004712	1659.2	0.040427	1782.7	0.075248	1870	0.113565	1945.2	0.151882	1996.6
0.005475	1664	0.041136	1785.3	0.076036	1872.1	0.114669	1946	0.152749	1997.9
0.006184	1669.2	0.041872	1787	0.076956	1875.1	0.115378	1946.9	0.153643	2000.5
0.007209	1676.5	0.042818	1790.1	0.077928	1875.1	0.116456	1948.6	0.15451	2001
0.008076	1681.7	0.043659	1793.5	0.079295	1876.9	0.117402	1949.1	0.155666	2002.3
0.00897	1685.1	0.044605	1794.4	0.080057	1879.5	0.118505	1951.2	0.156428	2003.1
0.009889	1689.4	0.045315	1795.7	0.08074	1881.6	0.119268	1952.1	0.157296	2003.1
0.010835	1693.8	0.046445	1800	0.081529	1882.9	0.120266	1953.4	0.158373	2004
0.011808	1698.5	0.047338	1800.9	0.08237	1884.7	0.121081	1954.2	0.159477	2006.6
0.012675	1702.4	0.048048	1803	0.083316	1887.2	0.122237	1956.8	0.16087	2007.9
0.013437	1705.8	0.048941	1805.6	0.084446	1889.8	0.123341	1958.1	0.162	2010.5
0.014199	1709.7	0.049756	1807.3	0.085287	1891.6	0.124471	1958.6	0.162604	2010.5
0.014961	1713.6	0.050492	1808.6	0.086128	1893.7	0.12526	1960.3	0.163971	2010.5
0.01575	1715.3	0.051569	1810.4	0.0876	1897.2	0.126232	1961.6	0.164838	2011.3
0.01638	1716.2	0.052332	1812.5	0.088335	1898.9	0.127204	1962.5	0.165863	2012.2
0.017064	1717.9	0.053173	1816	0.08936	1900.2	0.128098	1963.8	0.167361	2013.9
0.017852	1722.3	0.05383	1817.7	0.090438	1902.8	0.129202	1965.5	0.168255	2013.5
0.019114	1725.3	0.054618	1820.7	0.090858	1903.7	0.130279	1967.2	0.169306	2015.2
0.020086	1727.4	0.055748	1825	0.091726	1905	0.13091	1968.9	0.170252	2016.5
0.020874	1730.5	0.056799	1825.9	0.092619	1905.8	0.131725	1971.1	0.171067	2018.3
0.021584	1733.5	0.057456	1826.8	0.093513	1906.7	0.132618	1972.4	0.171881	2020.4
0.022504	1736.5	0.05835	1828.5	0.094538	1908.4	0.133538	1974.6	0.172591	2020.4
0.023266	1739.5	0.059086	1831.1	0.095773	1911.9	0.134353	1974.6	0.173564	2020
0.024159	1741.7	0.060137	1833.2	0.097218	1914	0.135167	1975.9	0.174378	2020.9
0.024869	1743.8	0.061057	1835	0.097823	1914.9	0.136035	1975.9	0.175745	2022.2
0.026051	1747.7	0.06195	1836.7	0.098795	1917.9	0.136797	1978	0.176691	2022.2
0.027234	1750.3	0.062712	1838.9	0.09961	1920.1	0.138216	1980.6	0.177479	2023.9
0.027996	1752	0.063842	1841	0.100398	1924.4	0.139083	1981.9	0.178136	2024.3
0.028653	1755.1	0.064815	1844.9	0.101292	1924.8	0.13995	1982.8	0.178583	2024.3
0.029678	1757.2	0.065524	1847.9	0.102054	1925.7	0.140844	1983.2		
0.030466	1759	0.066181	1849.7	0.102816	1926.1	0.141659	1984.9		
0.031229	1762	0.067154	1852.2	0.103657	1926.6	0.142631	1985.8		

Table A-3 Flow stress versus plastic strain of RHA for initial temperature 77 K and plastic strain rate 2500/s

Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)
0.019951	1791.9	0.042917	1817.4	0.065427	1787.8	0.088992	1780.1	0.116886	1739.6
0.020109	1794.6	0.043495	1816.8	0.065901	1789.4	0.089491	1778.7	0.117385	1738.7
0.02032	1797.1	0.043915	1816.8	0.066295	1789.4	0.090043	1777	0.117884	1737.8
0.020504	1801	0.044336	1816.9	0.066663	1790.8	0.090673	1775	0.118331	1737
0.020742	1805.7	0.044808	1813.9	0.066873	1791.2	0.091224	1774.4	0.118804	1737
0.021005	1809.9	0.045176	1812.8	0.067188	1790.2	0.091776	1772.6	0.119277	1736.9
0.021242	1813.7	0.045517	1810.5	0.067477	1788.1	0.092301	1769.7	0.119881	1737
0.021505	1817.4	0.045937	1808.5	0.068002	1785.1	0.092957	1768.1	0.120432	1734.1
0.021664	1820.1	0.046436	1806.3	0.06829	1783.8	0.093561	1767.4	0.12101	1731.9
0.021979	1823.8	0.046829	1801.9	0.068605	1782.4	0.094271	1768.1	0.121587	1730.9
0.022426	1824.6	0.047249	1797.4	0.069156	1778.5	0.09477	1769.2	0.122086	1729.2
0.022768	1824.8	0.047747	1792.4	0.069603	1777.3	0.095401	1769.2	0.122428	1728.2
0.023162	1826.6	0.048245	1788.5	0.070049	1774.9	0.0959	1771.1	0.123347	1727.4
0.02353	1826.4	0.048665	1784.9	0.070469	1774.9	0.096347	1772.6	0.123872	1726.4
0.024055	1827.1	0.049085	1780.8	0.070864	1775.8	0.097109	1773.6	0.124608	1725.5
0.024529	1828.3	0.049557	1777.1	0.071232	1776.8	0.097793	1773.9	0.125238	1726.3
0.024949	1828.4	0.050003	1773.8	0.071626	1778.9	0.098475	1772.3	0.125817	1727.2
0.025868	1827.9	0.050476	1772.1	0.0721	1782.1	0.099105	1768.8	0.126237	1728
0.026341	1828.7	0.051053	1769.8	0.072441	1783.1	0.099683	1767.3	0.126736	1728.6
0.026893	1827.7	0.051474	1769.1	0.072993	1785.7	0.100155	1764.9	0.127393	1727.5
0.027629	1828	0.051841	1769.6	0.073309	1788.7	0.100759	1763.1	0.127944	1726.1
0.028207	1827.7	0.052262	1769.3	0.073625	1791.4	0.101022	1762.2	0.128496	1722.9
0.028653	1827.3	0.052709	1771.8	0.073862	1792.5	0.101758	1762.4	0.129046	1718.8
0.029231	1827.1	0.053129	1773.5	0.074282	1793.4	0.102152	1763.3	0.129414	1715.9
0.02973	1826.6	0.053445	1774.9	0.074781	1794.1	0.102651	1764.7	0.129886	1711.3
0.030177	1825.7	0.053787	1777.2	0.075202	1794.5	0.103151	1765.4	0.130647	1707.3
0.030702	1823.9	0.054155	1780.2	0.075911	1794.4	0.103703	1768.3	0.131145	1703.4
0.031174	1821.4	0.05455	1784.4	0.076226	1793.4	0.104019	1770.5	0.131644	1701.4
0.031699	1819.5	0.054919	1788.3	0.077014	1792	0.104308	1771.5	0.132117	1701.3
0.032198	1818.2	0.055129	1789.9	0.077618	1790.5	0.104834	1774.8	0.132432	1700.4
0.032645	1818.6	0.055629	1793.9	0.078091	1790	0.105044	1775.6	0.1328	1699.7
0.03346	1819.2	0.056076	1797.6	0.078511	1789.2	0.105465	1778.4	0.133168	1699.9
0.033985	1822	0.056497	1800.9	0.079089	1788.2	0.105964	1779.8	0.133509	1699.6
0.034327	1824.3	0.056813	1803.2	0.079483	1787.9	0.10649	1780	0.133982	1700.3
0.034748	1827.3	0.057207	1806.2	0.079877	1787.1	0.107015	1778.3	0.134376	1701.2
0.035169	1829.8	0.057707	1808.9	0.080507	1785.6	0.107724	1775.8	0.134666	1701.8
0.035537	1831.6	0.058154	1809.7	0.081111	1784.5	0.108117	1773.2	0.13506	1702.2
0.035905	1833.5	0.058627	1808.9	0.081611	1784	0.108564	1771.2	0.1359	1701.5
0.036273	1834.9	0.059047	1807.7	0.082057	1783.9	0.109194	1767.3	0.136504	1697.8
0.036667	1833.7	0.059388	1806.2	0.082556	1783.4	0.109482	1764.9	0.137081	1692.7
0.037087	1832.5	0.059755	1803.5	0.08295	1783.5	0.109928	1760.2	0.137553	1687.3
0.03756	1830.8	0.06028	1800.5	0.083633	1783.1	0.110295	1756.4	0.137841	1681.6
0.038137	1827.2	0.060727	1798.5	0.084264	1782.7	0.110846	1751.9	0.138102	1675.8
0.03861	1824.6	0.061304	1794.1	0.084711	1783.5	0.111476	1748.1	0.138548	1669.8
0.039161	1822.3	0.06175	1791.3	0.08521	1783	0.112342	1746.2	0.139072	1662.8
0.039686	1818.9	0.062406	1787.6	0.085551	1783.5	0.112972	1743.5	0.139386	1658.2
0.040185	1817.9	0.063063	1786	0.086287	1784.2	0.11397	1741.7	0.139911	1652.8
0.040632	1817.5	0.06343	1786.2	0.086839	1783.2	0.114364	1741.4	0.140409	1648.5
0.041236	1818.1	0.063851	1787	0.087285	1783.2	0.115126	1741.2		
0.041682	1816.8	0.064403	1786.4	0.087889	1782.1	0.115573	1740.4		
0.042208	1816.3	0.065007	1786.9	0.088441	1781.1	0.116387	1739.7		

Table A-4 Flow stress versus plastic strain of RHA for initial temperature 298 K and plastic strain rate 0.001/s

Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)
0.000028	1064.6	0.031693	1298.4	0.068066	1346	0.103861	1362.6	0.1416	1374.4
0.000291	1081	0.032377	1299.3	0.069038	1346	0.104307	1362.6	0.142573	1375.7
0.000527	1094.4	0.033612	1302.3	0.069774	1346.9	0.105017	1362.2	0.144018	1375.7
0.000816	1107.3	0.034321	1304.1	0.070379	1346.5	0.105621	1361.7	0.144649	1375.3
0.001105	1120.7	0.035084	1305.8	0.071062	1346.9	0.1062	1361.7	0.145542	1375.3
0.001525	1135.3	0.03603	1307.1	0.071955	1347.3	0.106936	1362.2	0.14641	1374.4
0.001946	1140.9	0.036713	1309.3	0.072639	1346.9	0.107592	1362.2	0.147172	1376.6
0.00276	1152.2	0.037317	1310.1	0.073401	1348.2	0.108696	1362.6	0.148039	1376.6
0.003338	1159.5	0.038027	1311.8	0.073953	1348.2	0.109406	1363	0.148959	1376.6
0.00389	1166.4	0.03871	1312.3	0.074715	1347.4	0.110299	1363.9	0.149984	1378.3
0.004415	1173.3	0.039472	1313.1	0.075477	1348.7	0.110983	1365.6	0.150799	1379.2
0.005072	1180.2	0.040208	1314.4	0.076265	1350.4	0.111587	1365.2	0.151377	1378.7
0.005729	1185.8	0.041207	1317.9	0.077606	1350.4	0.112323	1364.8	0.153032	1377.4
0.00636	1192.3	0.041864	1317.5	0.078421	1350.4	0.113033	1364.8	0.154189	1376.2
0.007148	1197.4	0.042784	1319.2	0.079104	1350	0.113663	1365.2	0.154846	1377
0.007858	1203.1	0.04352	1321.4	0.079629	1350.8	0.114452	1365.6	0.155582	1377.5
0.008462	1206.5	0.044203	1320.9	0.080418	1351.3	0.115661	1364.4	0.156554	1377.5
0.009014	1210.8	0.044755	1320.9	0.081233	1351.7	0.116791	1365.7	0.157264	1378.3
0.009592	1215.6	0.045543	1323.1	0.081679	1352.6	0.117658	1365.2	0.158236	1379.2
0.010197	1218.2	0.046594	1325.7	0.082415	1352.6	0.118447	1365.2	0.159103	1378.8
0.010749	1220.7	0.047173	1326.1	0.082993	1353.4	0.119288	1365.7	0.159734	1379.6
0.011353	1225.9	0.048014	1326.5	0.083598	1353	0.119997	1366.5	0.160654	1378.8
0.011957	1228.9	0.048986	1328.3	0.084202	1354.3	0.120733	1368.3	0.161732	1378.3
0.012641	1232	0.049879	1329.1	0.085122	1354.7	0.121653	1368.7	0.162625	1378.4
0.013376	1236.3	0.050747	1330.9	0.086095	1354.7	0.122573	1367	0.163335	1378.4
0.014664	1241	0.05164	1332.2	0.086699	1355.6	0.124255	1367.4	0.164255	1380.1
0.015426	1246.6	0.052297	1331.7	0.08754	1356	0.124938	1366.5	0.165201	1381.4
0.016346	1249.7	0.052954	1332.2	0.088145	1356.5	0.125937	1365.3	0.165936	1381.8
0.016977	1252.3	0.053717	1331.7	0.088959	1356.9	0.126489	1365.7	0.166593	1381.8
0.017581	1254.8	0.054531	1332.6	0.089721	1358.6	0.127119	1366.1	0.167303	1380.5
0.018317	1258.3	0.055293	1333	0.090352	1359.1	0.127855	1366.1	0.169169	1381
0.019158	1260.9	0.056187	1334.8	0.091167	1358.2	0.12867	1367.9	0.169958	1381
0.019841	1263.9	0.056713	1334.3	0.091666	1359.1	0.129038	1367.9	0.170825	1381.4
0.02063	1265.6	0.057527	1335.6	0.092849	1358.6	0.129406	1368.3	0.171587	1381
0.021365	1269.1	0.058684	1337.4	0.09348	1357.4	0.130352	1369.2	0.172375	1381.4
0.022338	1271.2	0.05963	1339.1	0.094531	1358.7	0.131035	1369.6	0.173348	1382.3
0.023179	1274.3	0.060392	1338.7	0.095319	1356.9	0.132165	1370.5	0.17453	1382.3
0.023941	1277.3	0.061233	1339.1	0.096187	1359.5	0.133138	1370.9	0.175582	1381.4
0.025018	1281.2	0.061916	1340	0.097317	1359.1	0.134793	1371.8	0.176607	1383.2
0.025938	1283.8	0.062442	1340	0.098552	1359.5	0.135687	1371.3	0.177421	1383.2
0.026726	1285.5	0.063178	1340.4	0.099656	1360	0.136922	1372.2	0.178262	1384
0.027567	1287.2	0.063913	1341.3	0.100628	1360	0.1375	1373.5	0.179182	1384.5
0.028697	1291.5	0.064754	1341.7	0.10097	1360	0.138446	1373.5	0.180023	1384.5
0.029696	1293.3	0.065517	1341.7	0.101732	1360.4	0.139314	1373.5		
0.030511	1295.4	0.066226	1342.6	0.102468	1360.8	0.140155	1374.4		
0.031089	1297.6	0.066857	1342.2	0.103177	1362.1	0.140786	1374.8		

Table A-5 Flow stress versus plastic strain of RHA for initial temperature 298 K and plastic strain rate 0.1/s

Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)
0.000212	1068.9	0.028986	1310.9	0.067382	1363.7	0.104517	1381.6	0.14499	1398.1
0.000475	1087.9	0.029801	1310.9	0.06825	1364.1	0.105017	1382	0.145489	1397.7
0.000606	1099.5	0.030458	1312.7	0.069196	1365	0.106173	1383.7	0.146698	1397.7
0.000842	1116.4	0.031036	1315.3	0.070037	1366.3	0.106856	1383.7	0.147723	1398.1
0.000973	1133.2	0.031824	1316.6	0.07093	1368.5	0.107592	1383.7	0.148275	1398.6
0.001315	1149.1	0.032955	1319.2	0.071771	1368.5	0.108617	1384.2	0.148827	1399
0.001604	1159.9	0.033769	1320	0.072744	1369.3	0.109905	1384.6	0.149642	1400.3
0.001945	1171.6	0.034453	1320.9	0.073795	1370.2	0.111035	1385.5	0.150351	1398.6
0.002339	1178.9	0.035819	1324.3	0.074741	1370.2	0.111692	1384.2	0.151587	1399
0.002681	1185.4	0.036397	1324.3	0.07574	1371.1	0.112481	1385	0.152769	1398.1
0.003259	1193.1	0.036897	1326.1	0.076502	1372.4	0.113348	1385.9	0.153715	1398.1
0.0036	1195.7	0.038184	1330.4	0.077605	1373.7	0.114268	1386.8	0.154609	1399
0.004415	1206.5	0.039183	1333.4	0.078499	1373.2	0.115082	1387.2	0.155135	1399
0.005151	1211.2	0.039787	1333.8	0.079445	1373.7	0.116055	1388.1	0.155897	1399.5
0.005703	1216.8	0.040523	1335.1	0.080286	1373.3	0.116817	1387.6	0.156344	1400.3
0.006281	1221.2	0.0421	1336.9	0.08118	1374.1	0.117842	1388.5	0.157447	1399.9
0.006754	1224.6	0.043046	1337.7	0.082205	1375.8	0.118551	1387.2	0.158499	1401.2
0.007411	1230.2	0.043624	1339	0.083177	1374.6	0.119471	1387.2	0.158919	1400.8
0.00791	1234.1	0.044571	1340.3	0.084176	1374.6	0.120391	1386.4	0.159313	1400.3
0.008567	1238.4	0.04599	1341.2	0.085017	1375.4	0.121127	1385.5	0.160916	1400.3
0.009119	1243.2	0.046515	1340.8	0.0857	1376.3	0.121863	1387.2	0.16181	1401.2
0.009671	1246.2	0.047251	1342.5	0.086462	1376.3	0.122861	1386.8	0.162414	1401.6
0.010222	1250.1	0.047856	1342.5	0.087145	1376.7	0.124018	1386.4	0.163623	1402.9
0.011431	1254.8	0.048539	1344.7	0.087934	1376.7	0.124649	1388.5	0.164517	1402.5
0.012299	1257.8	0.049722	1345.5	0.088775	1377.2	0.125411	1388.5	0.165595	1402.5
0.012982	1261.3	0.050352	1346.8	0.089642	1376.7	0.126698	1390.7	0.166383	1402.9
0.013691	1263.4	0.051272	1349.4	0.090142	1377.2	0.127671	1389.8	0.167355	1403.4
0.014322	1265.6	0.051903	1349.4	0.091061	1378.5	0.128459	1390.7	0.168223	1404.2
0.01511	1266.9	0.052639	1351.1	0.091902	1379.3	0.129484	1391.6	0.169195	1403.8
0.01603	1270.8	0.053427	1350.7	0.092796	1378.5	0.130772	1391.6	0.169931	1403.8
0.016766	1273.8	0.054216	1352.9	0.093689	1378.9	0.131193	1391.6	0.17093	1403.8
0.017528	1277.7	0.055162	1352.4	0.094636	1378.5	0.132112	1392.4	0.172007	1404.3
0.018763	1282	0.056029	1355.5	0.095214	1378.9	0.132848	1392.9	0.172795	1405.1
0.019815	1285	0.056686	1356.8	0.095897	1378.5	0.134451	1393.8	0.173558	1404.3
0.020393	1286.8	0.057947	1357.6	0.096659	1378.9	0.135292	1393.3	0.174372	1404.7
0.021391	1287.6	0.058736	1359.4	0.097264	1380.2	0.136659	1395.1	0.175318	1405.1
0.022311	1291.9	0.059656	1359.4	0.098026	1380.2	0.13771	1394.2	0.175897	1405.6
0.023231	1296.3	0.060549	1360.2	0.099209	1382	0.13863	1394.6	0.17679	1406.4
0.023967	1298.8	0.061679	1359.8	0.100102	1379.4	0.13955	1395.1	0.177657	1406
0.024703	1298.9	0.062573	1360.7	0.100864	1380.2	0.140259	1395.5	0.178446	1406.4
0.025859	1302.3	0.063519	1359.8	0.101889	1381.1	0.141153	1395.9	0.179155	1406.9
0.026595	1304.5	0.064307	1362.8	0.102599	1382	0.142204	1397.2	0.179786	1406.9
0.027672	1306.2	0.065332	1363.7	0.103282	1381.1	0.142809	1396.8	0.180233	1408.2
0.028277	1310.1	0.066278	1362.8	0.103887	1380.7	0.144149	1397.7		

Table A-6 Flow stress versus plastic strain of RHA for initial temperature 298 K and plastic strain rate 3500/s

Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)
0.028009	1290.9	0.057762	1379.4	0.093368	1429.5	0.123083	1441.7	0.158363	1430.6
0.02843	1295	0.058157	1380.5	0.09371	1429.8	0.123398	1442.7	0.158626	1433.9
0.028772	1298.3	0.05863	1381.4	0.094235	1431	0.123793	1444.4	0.158916	1437.9
0.029141	1302.1	0.059024	1382.1	0.094787	1432.6	0.124187	1445.8	0.1591	1439.5
0.02943	1305.9	0.059418	1383.6	0.095339	1434.1	0.125448	1444.6	0.159521	1443.7
0.029694	1309.3	0.059944	1386.1	0.096023	1436.3	0.125868	1445.9	0.159916	1446.8
0.030088	1310.4	0.060759	1389	0.096469	1436.4	0.127365	1442.9	0.160232	1448.5
0.030535	1311.5	0.061573	1389.1	0.097389	1438.6	0.127733	1441.6	0.160941	1451.2
0.031112	1309.5	0.062204	1389.4	0.098335	1438.7	0.128337	1439.2	0.161441	1452.3
0.03148	1307.7	0.063386	1390.1	0.098676	1437.8	0.129098	1436.2	0.161887	1451.2
0.0319	1307.2	0.063885	1389.8	0.099307	1437.3	0.129702	1435.3	0.162517	1448.1
0.03232	1306.6	0.064279	1390.5	0.099753	1435.1	0.130227	1433	0.163147	1444.2
0.032662	1306.1	0.064936	1390.4	0.100278	1432.7	0.130962	1430.8	0.163567	1440.7
0.033318	1306.9	0.06554	1390.7	0.100803	1429.2	0.131382	1429	0.164039	1436.3
0.033765	1306.7	0.066355	1390.8	0.101721	1420.8	0.131986	1427.4	0.164327	1432.5
0.034264	1308.2	0.067143	1391.3	0.10193	1418.6	0.132459	1426.3	0.164825	1428.9
0.034659	1308.9	0.067695	1390.9	0.10235	1414.8	0.133693	1421.5	0.165324	1425.6
0.035079	1310	0.068457	1391.4	0.102874	1409.4	0.134796	1422.5	0.165822	1422.3
0.035447	1311.2	0.069087	1391.7	0.10332	1404	0.135716	1425.3	0.166295	1421.9
0.035868	1312.3	0.069639	1392.5	0.103845	1399.8	0.136505	1428.3	0.16682	1420.8
0.036577	1313.8	0.070375	1393.9	0.104396	1398.6	0.136926	1430.7	0.167346	1419.7
0.037181	1313.4	0.0709	1395.2	0.105027	1401.2	0.137294	1434	0.168055	1419.4
0.037864	1313.7	0.071347	1396.1	0.1055	1402.8	0.137557	1436.2	0.16858	1420.1
0.038469	1314.6	0.072083	1397.5	0.105764	1406.7	0.138004	1439.3	0.168948	1420.3
0.039126	1315.8	0.07274	1398.6	0.106264	1413	0.138504	1443.1	0.1695	1420.9
0.039625	1315.9	0.07337	1398.9	0.10658	1416.9	0.138689	1444.7	0.17021	1422.1
0.04015	1316.4	0.074054	1400.3	0.106896	1421.8	0.138925	1447	0.170709	1423.2
0.040728	1316.9	0.074527	1400.4	0.107476	1430.7	0.139372	1449.8	0.171549	1422
0.041096	1318.5	0.075105	1402	0.107634	1432.8	0.139767	1451.4	0.171917	1422.5
0.0422	1321.6	0.075736	1403.9	0.107818	1435.7	0.140293	1454	0.172364	1421.9
0.04249	1323.7	0.076156	1404.1	0.108003	1438.9	0.14116	1457.2	0.172915	1418.9
0.042937	1325.5	0.076603	1405.4	0.108292	1442.4	0.142184	1453.8	0.173308	1416.1
0.043331	1328.3	0.077102	1407.1	0.108845	1447.4	0.142815	1452.9	0.173728	1413
0.043778	1331.4	0.077628	1408	0.109108	1450.9	0.143287	1449.2	0.174306	1410.7
0.044146	1333.4	0.078337	1408.9	0.109714	1456.6	0.14368	1447.5	0.174726	1409.7
0.044646	1335.2	0.078811	1410.7	0.110029	1459.6	0.1441	1443.5	0.175172	1409
0.045145	1337.8	0.079047	1411.9	0.110371	1460.6	0.144651	1439.7	0.175698	1411.2
0.045593	1341.4	0.079416	1415	0.111028	1462.7	0.145229	1437.7	0.175909	1412.3
0.046223	1343.8	0.079705	1417.4	0.111527	1460.8	0.145649	1435.9	0.176356	1415.3
0.046539	1346.3	0.080152	1420.7	0.11221	1459.4	0.146174	1433.7	0.17654	1418
0.046934	1349	0.080547	1422.9	0.112656	1455.6	0.146646	1431.8	0.176777	1420.5
0.047407	1351.2	0.080994	1424.7	0.113128	1450.7	0.147093	1431.6	0.177041	1427
0.047985	1354.2	0.081493	1427.5	0.113521	1446.2	0.148353	1429	0.177226	1431.8
0.048275	1357.1	0.081861	1429	0.114019	1439.3	0.149246	1426.2	0.177411	1435.9
0.048722	1359.7	0.082361	1430.7	0.114465	1435.2	0.150244	1425.1	0.177753	1441.6
0.049011	1361.8	0.082939	1432.2	0.114937	1430.6	0.150822	1423.5	0.178043	1445.5
0.04951	1362.3	0.083938	1433.9	0.115435	1425.6	0.151321	1421.6	0.178307	1449.5
0.050088	1362.8	0.084647	1433.3	0.115881	1421.5	0.152029	1419.9	0.178518	1452.8
0.05043	1365.2	0.085303	1432.3	0.116327	1417.4	0.152135	1420.1	0.178597	1454.8
0.050877	1366.2	0.085934	1432.6	0.117298	1412.5	0.153263	1414	0.178887	1459.1
0.051403	1369.6	0.086669	1430.7	0.118348	1412.1	0.153867	1411.6	0.179308	1465.7
0.052191	1369.7	0.087352	1430.5	0.118769	1413	0.154392	1410.5	0.179624	1468.5
0.052822	1370.4	0.087851	1430.5	0.119426	1415.6	0.154838	1408	0.180255	1470.4
0.053374	1372.1	0.088508	1429.2	0.119768	1418.1	0.155574	1408.8	0.180676	1472
0.053847	1372.7	0.089007	1428.8	0.120031	1419.7	0.155784	1409.6	0.18107	1472.1
0.054477	1372.6	0.090031	1428.1	0.120558	1424.7	0.15631	1411.4	0.1817	1469.7
0.054977	1373.5	0.090793	1427.8	0.121057	1427.4	0.156494	1412.6	0.182277	1466.3
0.055423	1374.5	0.09145	1427.6	0.121426	1430.3	0.157125	1415.8	0.182775	1460.5
0.055975	1375.8	0.091739	1428.4	0.121899	1433.4	0.157467	1418.6	0.183116	1456.8
0.056737	1376	0.092265	1428.8	0.122346	1436.6	0.157783	1421.8	0.183351	1448.8
0.057263	1377.3	0.092921	1428.6	0.122846	1440.5	0.157941	1423.8	0.183586	1443.4

Table A-7 Flow stress versus plastic strain of RHA for initial temperature 298 K and plastic strain rate 7000/s

Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)
0.03648	1359	0.058189	1417.2	0.08481	1462.9	0.116208	1484.7	0.154932	1485
0.036769	1360.7	0.058662	1417.8	0.08531	1465.7	0.116418	1485.5	0.155746	1483
0.037032	1364.1	0.058925	1416.6	0.085862	1468.5	0.117022	1485.7	0.156297	1479.6
0.037427	1366.5	0.059424	1416.2	0.086177	1470.4	0.117601	1486	0.157164	1477.6
0.037821	1369.3	0.059975	1415.4	0.086546	1472.6	0.118179	1487.8	0.157689	1477.5
0.038216	1372.5	0.060474	1414.7	0.087072	1476.6	0.119362	1491.8	0.158293	1477.7
0.03869	1377.2	0.061079	1414.2	0.087466	1479.1	0.119913	1491.3	0.159003	1477.4
0.039111	1380.4	0.061499	1413.2	0.087861	1480.9	0.120334	1491.8	0.159397	1477.1
0.039532	1383.9	0.061945	1412.6	0.088124	1482.9	0.120991	1492.4	0.159764	1477.7
0.039874	1387.2	0.062313	1412.4	0.088623	1484.9	0.121516	1493.1	0.16029	1479.1
0.040268	1390.4	0.063154	1411.6	0.089359	1488.5	0.121989	1493.5	0.160869	1481.5
0.040715	1393.9	0.06381	1412.7	0.0902	1490.4	0.122646	1494.4	0.161184	1482.2
0.041136	1396.6	0.064284	1414	0.090779	1491.9	0.123356	1494.5	0.16171	1483.9
0.0414	1400.4	0.065046	1415.8	0.091567	1492.7	0.124643	1494.4	0.162183	1485.4
0.041715	1404.1	0.06544	1417.2	0.091882	1491.8	0.125588	1492.6	0.162788	1487.8
0.042005	1407.4	0.065808	1419.5	0.092565	1491.5	0.126376	1492.9	0.163602	1489.2
0.042347	1411.1	0.06615	1420.7	0.093011	1489.3	0.126954	1492.4	0.164181	1490.5
0.042742	1413	0.066518	1421.7	0.093458	1487.8	0.127453	1491.3	0.164943	1491.2
0.043084	1415.4	0.066807	1422.5	0.093983	1485.8	0.128267	1490.4	0.166178	1493.7
0.043636	1420.4	0.067307	1425	0.094586	1483.7	0.128767	1490.8	0.166414	1492.2
0.044031	1424	0.068043	1428	0.095059	1482.6	0.129213	1489.7	0.167228	1491.3
0.044768	1432	0.068437	1429.9	0.095374	1480.2	0.129844	1489.9	0.16778	1489.8
0.045215	1434.6	0.0687	1431.1	0.095794	1477.5	0.13079	1490.7	0.168699	1490.2
0.045557	1437.9	0.069147	1432.9	0.096476	1474.9	0.13121	1491.6	0.169198	1488.3
0.045899	1440.3	0.069462	1434	0.097054	1472.1	0.13234	1493.3	0.169933	1486.1
0.046267	1441.4	0.069857	1435.9	0.097868	1468.5	0.133812	1494.8	0.170458	1484.2
0.046766	1443.9	0.070199	1437	0.098419	1466.9	0.1346	1494.7	0.171246	1480.4
0.047134	1444.2	0.070593	1438	0.098682	1466.1	0.135204	1493.4	0.171824	1479.6
0.047554	1443.2	0.071066	1439.3	0.09918	1463.4	0.135651	1494.3	0.172296	1477
0.048105	1439.9	0.071749	1440.7	0.099758	1462.5	0.136202	1493.8	0.172847	1474.4
0.048499	1437.2	0.072143	1440.9	0.100467	1462.2	0.136727	1493.4	0.173451	1471.3
0.048761	1432.7	0.072853	1441.9	0.100887	1462	0.137253	1493	0.174028	1468.6
0.049207	1428.3	0.073457	1441.4	0.101387	1460.9	0.137831	1493.6	0.174632	1466.9
0.049548	1424.5	0.074271	1440.2	0.101859	1461.3	0.139013	1493.6	0.175183	1464.7
0.049888	1420.3	0.074718	1439.5	0.102622	1462.1	0.139407	1493.8	0.175866	1464.4
0.050203	1416.6	0.075742	1438	0.103383	1461.8	0.140432	1493.4	0.176864	1462.1
0.050544	1412.4	0.076267	1437.2	0.104119	1463.8	0.141193	1492.2	0.177521	1461.9
0.050832	1408.2	0.076687	1436.6	0.104513	1464	0.141771	1490.6	0.177915	1460.9
0.051173	1403.6	0.077265	1436.2	0.105722	1465.7	0.142349	1490.4	0.178466	1459.4
0.051671	1398.8	0.078027	1435.8	0.10609	1465.9	0.142848	1491.5	0.178887	1459.1
0.052091	1397.4	0.078579	1435.4	0.106537	1466	0.143505	1490.5	0.179464	1458.6
0.052537	1395.9	0.078841	1435.4	0.107299	1466.4	0.144372	1489.6	0.180147	1457.2
0.053089	1394.3	0.079314	1435.1	0.107588	1466.7	0.145423	1490.6	0.18104	1453.4
0.053457	1394.6	0.079761	1436.5	0.108113	1467.5	0.146158	1490.6	0.181512	1450.8
0.053904	1396.4	0.080024	1437.3	0.108691	1468.6	0.146605	1490.3	0.182115	1446.3
0.054193	1398.9	0.080313	1437.7	0.1099	1471	0.147735	1491.5	0.182798	1442.8
0.054798	1401.3	0.080707	1438.7	0.1104	1473	0.148444	1492.3	0.183349	1439.5
0.055166	1404	0.080996	1440.3	0.111451	1474.9	0.149101	1493.2	0.183899	1433.6
0.055482	1406	0.081549	1444.6	0.112371	1475.8	0.149758	1492.2	0.184476	1428.8
0.055823	1408.4	0.082154	1446.5	0.11287	1477	0.150415	1492.3	0.184975	1424.8
0.056244	1410.2	0.082863	1449.9	0.113527	1477.5	0.151439	1492.6	0.185578	1420.3
0.056665	1411.7	0.083153	1451.8	0.114	1479.1	0.151912	1491.5		
0.057059	1413.6	0.083495	1453.7	0.114526	1480.7	0.152779	1489.4		
0.057453	1415	0.083968	1458.2	0.11513	1482.1	0.153619	1487.4		
0.057795	1416.6	0.084494	1460.6	0.115525	1483	0.15417	1486.2		

Table A-8 Flow stress versus plastic strain of RHA for initial temperature 473 K and plastic strain rate 3000/s

Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)
0.030984	1177.8	0.05687	1235.8	0.087142	1287	0.118271	1280.2	0.148874	1270.6
0.031406	1185.6	0.057501	1237.4	0.087667	1286.7	0.118928	1280.9	0.149452	1271.4
0.031722	1192	0.058105	1240.3	0.088193	1286.4	0.119769	1281.7	0.149873	1272.3
0.032065	1200.8	0.05871	1244.5	0.088718	1285.3	0.120347	1282.4	0.150503	1272.6
0.03246	1205.3	0.058974	1247	0.089243	1285	0.120846	1281.8	0.150871	1274
0.032829	1209	0.0595	1251.6	0.089742	1281.6	0.121424	1281.8	0.151581	1276
0.033407	1211.7	0.059895	1256	0.091055	1278.7	0.122028	1282.5	0.151897	1276.7
0.033986	1214.7	0.060421	1259	0.092158	1277.2	0.122501	1283.1	0.152265	1277.7
0.034328	1217.6	0.060841	1262.1	0.092579	1279	0.123132	1284.5	0.152843	1280.3
0.034774	1219.6	0.061367	1265.1	0.093472	1281.8	0.1235	1285.2	0.153737	1283.2
0.0353	1221.8	0.062392	1265.8	0.094051	1283.8	0.124315	1287.5	0.154736	1286
0.035852	1224.1	0.06276	1266.9	0.09476	1285.6	0.124788	1289.2	0.155708	1287.3
0.036509	1226.2	0.063102	1267.3	0.09497	1286.1	0.124867	1289.1	0.155997	1287.7
0.037088	1227.5	0.063889	1266.3	0.095707	1289.5	0.125366	1290.7	0.156444	1287.8
0.037744	1228.8	0.064625	1265.3	0.096259	1290.7	0.126206	1284.6	0.156785	1287.7
0.038165	1229.9	0.065097	1263.5	0.096653	1292.1	0.126888	1281.5	0.157573	1286.7
0.038717	1230.9	0.065754	1262.2	0.097047	1292.4	0.127492	1278.4	0.158203	1284.4
0.039479	1233.6	0.066279	1261.5	0.097546	1291	0.128043	1277.3	0.158833	1281.7
0.040031	1233.4	0.066647	1260.2	0.098203	1290.9	0.128831	1275.5	0.159463	1279.8
0.04053	1234.8	0.067251	1260.2	0.098597	1291.5	0.129356	1274.8	0.159988	1276.5
0.04103	1237.5	0.067671	1259.7	0.099306	1289.4	0.129645	1274.8	0.16046	1272.9
0.041529	1238.9	0.068302	1261.3	0.100172	1288.2	0.130355	1275.7	0.160854	1270.9
0.042081	1240.7	0.068722	1262.4	0.101196	1286.1	0.130749	1276	0.161457	1266.4
0.042686	1243.3	0.069143	1264.3	0.102405	1285.7	0.131143	1276.2	0.162034	1263.8
0.043264	1245	0.069643	1267.2	0.10293	1285.8	0.131458	1277.3	0.162612	1261.3
0.043764	1248.1	0.070563	1269.7	0.103298	1285.3	0.132247	1281.2	0.163085	1260.3
0.044316	1250.7	0.070799	1271	0.103902	1285.2	0.132589	1282.3	0.163767	1259.4
0.044999	1252.7	0.071641	1274.7	0.104375	1284.2	0.133062	1285.8	0.16424	1260.2
0.045525	1254.9	0.072114	1277.7	0.104821	1284	0.133404	1288	0.165134	1261.6
0.046103	1256.2	0.072482	1278.8	0.105373	1284.1	0.133851	1289.3	0.166081	1267.4
0.046734	1257.5	0.073113	1281.6	0.10582	1283.9	0.134456	1291.2	0.166712	1270.6
0.047076	1259.1	0.073586	1283.4	0.106266	1284.8	0.13527	1290.1	0.16758	1274.2
0.047523	1261.8	0.073928	1283.8	0.106897	1286.7	0.135953	1290.3	0.167921	1276
0.047943	1263.8	0.074821	1285.8	0.107187	1287.9	0.137004	1289.7	0.16829	1278.1
0.048364	1265.3	0.075767	1287	0.10766	1291.2	0.137686	1285.4	0.169131	1283.2
0.048837	1267.1	0.076686	1284.9	0.107923	1292.8	0.138421	1284	0.169446	1283.1
0.049284	1268.2	0.077737	1283.9	0.108528	1295.5	0.139209	1283	0.170103	1284.1
0.049809	1269.6	0.078341	1282.7	0.109867	1292.1	0.13976	1280.8	0.170629	1283.4
0.050283	1271.8	0.078787	1280.5	0.110261	1290.4	0.140469	1279.8	0.171548	1280.3
0.050598	1272.7	0.079575	1279.5	0.110812	1288.1	0.140942	1278.1	0.172545	1277.9
0.051071	1272.4	0.080284	1277.8	0.111337	1285.1	0.14152	1276.9	0.172992	1276.6
0.051622	1270.5	0.080993	1276.4	0.112072	1283	0.141888	1277.2	0.173516	1273
0.052147	1266.9	0.08165	1275.2	0.112518	1282	0.14257	1275.9	0.174199	1269.9
0.05275	1260.4	0.082044	1275.1	0.113227	1279.1	0.14328	1275.7	0.175354	1264.4
0.053301	1257.2	0.082858	1275.2	0.113936	1279.4	0.143726	1275.2	0.17601	1260.3
0.053878	1251.9	0.083515	1276.3	0.114436	1279.9	0.144304	1274	0.176901	1253.3
0.05435	1246.4	0.084172	1277.4	0.114882	1280.8	0.145013	1272.3	0.177478	1246.1
0.054795	1240.1	0.084671	1280	0.115539	1280.7	0.146116	1272.4	0.178002	1239.2
0.055215	1236.7	0.085592	1285.1	0.11588	1279.9	0.146878	1271.8	0.178657	1232.2
0.05574	1234.8	0.086433	1286.4	0.116695	1280.3	0.147535	1271.3	0.179181	1224
0.056344	1234.8	0.086853	1287	0.117457	1279.3	0.148139	1270.9		

Table A-9 Flow stress versus plastic strain of RHA for initial temperature 673 K and plastic strain rate 3000/s

Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)
0.020894	1007.3	0.045945	1103.7	0.077839	1123.5	0.117924	1113.9	0.155884	1111.3
0.021262	1009.8	0.046575	1102.6	0.07868	1124.1	0.118791	1112.9	0.156435	1112.2
0.021499	1013	0.047126	1100.3	0.079547	1126.3	0.119579	1110.1	0.156961	1112.7
0.021736	1016.6	0.047625	1099.3	0.079757	1126	0.120261	1108.5	0.157512	1112.1
0.022052	1020.5	0.048124	1098.8	0.080755	1123.7	0.120944	1107	0.158064	1111.4
0.022395	1025.2	0.048571	1099.5	0.081596	1122.7	0.121496	1107.9	0.158616	1111.9
0.02271	1028.2	0.049176	1102.1	0.082541	1120.4	0.1221	1108.3	0.159535	1110.9
0.023078	1030.8	0.049754	1102.3	0.082961	1120.3	0.122915	1109.7	0.160506	1108.3
0.02342	1032.1	0.050463	1103.1	0.083355	1118.3	0.123572	1110.5	0.160717	1108.4
0.023788	1033	0.050963	1105.4	0.084012	1118.3	0.123888	1112.8	0.16119	1109.3
0.024234	1031.7	0.051752	1111.5	0.085824	1116.6	0.124282	1114.6	0.161716	1112.4
0.024812	1028.1	0.05212	1113.5	0.086849	1121	0.125097	1118.7	0.162137	1116.7
0.025258	1023.8	0.052619	1114.2	0.087427	1119.5	0.12557	1120.4	0.162953	1125
0.025808	1017.7	0.053145	1116.1	0.08811	1121.9	0.126517	1123.9	0.163295	1128.2
0.026333	1012.1	0.053618	1117.2	0.088925	1125.3	0.127016	1126	0.163716	1132.9
0.026857	1007.3	0.054038	1118.7	0.089293	1125.7	0.128014	1126.3	0.164163	1136.1
0.027461	1002.5	0.054853	1119.8	0.089635	1126.4	0.128881	1126.8	0.164873	1139.6
0.027933	1000.7	0.05551	1120.3	0.090134	1128.2	0.129564	1123	0.165713	1135.3
0.02838	1002.8	0.056166	1119.1	0.09087	1130.1	0.130115	1119.3	0.166369	1130.2
0.028748	1004.9	0.056797	1118	0.091475	1132.2	0.130771	1116.7	0.166815	1125.2
0.029116	1008.3	0.057506	1115.9	0.092132	1135.8	0.131453	1113.6	0.167313	1119.9
0.029485	1013.8	0.058136	1113.6	0.092894	1135.7	0.132136	1112.4	0.167574	1111.9
0.029933	1018.8	0.058713	1110.9	0.093997	1133.2	0.132582	1111.2	0.168229	1102.4
0.030117	1023.3	0.059107	1107.9	0.094968	1128.5	0.133108	1111.7	0.168648	1096.8
0.030328	1028.5	0.059657	1103.6	0.095466	1125.6	0.133712	1113.3	0.169146	1091.4
0.030592	1033.7	0.060104	1101.9	0.096149	1121.2	0.134212	1115.7	0.169828	1087.7
0.030829	1038.9	0.060944	1098.9	0.096778	1117	0.134554	1117.6	0.170484	1086.3
0.03104	1042	0.061363	1095.6	0.097618	1114.4	0.135027	1120.5	0.171089	1088.9
0.031356	1047.5	0.062046	1092.8	0.098301	1113.6	0.135816	1121.8	0.171563	1092.7
0.031724	1051.3	0.062781	1091.5	0.098958	1115.6	0.136525	1124.4	0.172168	1099.7
0.03204	1054.8	0.063333	1089.3	0.099589	1118.8	0.137314	1126.9	0.172617	1108.3
0.032382	1059	0.063779	1088.4	0.100326	1121.9	0.137865	1125.1	0.172775	1111.8
0.032724	1062	0.064173	1088.3	0.100826	1126.8	0.138574	1122.4	0.173355	1122
0.033093	1064.1	0.064856	1087.9	0.101326	1132.1	0.139545	1118.2	0.174119	1134.6
0.033644	1064.7	0.065408	1089.4	0.101852	1136.1	0.140227	1111.1	0.174383	1138.3
0.034275	1065.7	0.065671	1090.7	0.102561	1136.5	0.141171	1105.1	0.175146	1146.1
0.034853	1066.3	0.066302	1094	0.103192	1138.5	0.141828	1103.9	0.17533	1145.9
0.035405	1067.8	0.066538	1095	0.104085	1139	0.142458	1103.6	0.176144	1144.6
0.035904	1068.1	0.066801	1096.7	0.10482	1135.8	0.142879	1105	0.176878	1133.9
0.036482	1070.8	0.067327	1099.3	0.105608	1133.3	0.143668	1109	0.177244	1125.9
0.037139	1071.7	0.067774	1101.7	0.106106	1130.7	0.14422	1112.1	0.177558	1119.7
0.037691	1072.8	0.068142	1102.9	0.106788	1124.1	0.144957	1118.7	0.178108	1110.7
0.038033	1074.5	0.068484	1103.7	0.107418	1117.9	0.145352	1122.8	0.178553	1103.3
0.038533	1077.7	0.068825	1104.1	0.108205	1112.4	0.145799	1126	0.179156	1097.9
0.03898	1080.9	0.069903	1106.5	0.108861	1110.8	0.14643	1130.5	0.179733	1094
0.039401	1084.2	0.070848	1105.4	0.109491	1109.7	0.146956	1132.9	0.180206	1093.9
0.0399	1087	0.07203	1104.9	0.110148	1109.7	0.147612	1129.9	0.180758	1095.8
0.040216	1089.9	0.072556	1103.9	0.11091	1110.4	0.148584	1128.4	0.1811	1097.5
0.040742	1093.5	0.073107	1103.6	0.111436	1112.1	0.149266	1122.7	0.181495	1101.8
0.041163	1096.8	0.073449	1103.6	0.11183	1113.6	0.149764	1118.4	0.18189	1107.1
0.041504	1097.6	0.074079	1104.5	0.112698	1117.6	0.150367	1114	0.182232	1113.2
0.041846	1099.7	0.074657	1105.9	0.113276	1119.3	0.150945	1110.3	0.182522	1117.2
0.042293	1101.7	0.074999	1107.9	0.113959	1121.2	0.151863	1105.6	0.182917	1122.9
0.042661	1103	0.075394	1109.5	0.114406	1121.8	0.152414	1102.3	0.183154	1123.7
0.043161	1104.5	0.075815	1112.6	0.115036	1121.1	0.153124	1103	0.183521	1122.2
0.04366	1106	0.076682	1116.4	0.115798	1120.2	0.153912	1104.3	0.184125	1118.3
0.044212	1107	0.07684	1117	0.116244	1119.7	0.154228	1106.2	0.184623	1110.8
0.044816	1107.5	0.077313	1120.1	0.116822	1117.9	0.154911	1108	0.184937	1105.8
0.04542	1105.6	0.077497	1120.7	0.117531	1115.1	0.15541	1110.4		

Table A-10 Flow stress versus plastic strain of RHA for initial temperature 873 K and plastic strain rate 3500/s

Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)	Strain	Stress (MPa)
0.024813	728.8	0.048262	834.8	0.08079	880.2	0.116274	859.3	0.151119	925.7
0.025024	734.6	0.048892	833.8	0.081551	879.5	0.116852	861.1	0.151592	926.4
0.025314	737.7	0.049286	833.9	0.08205	876.3	0.117694	864.2	0.152643	928.7
0.025577	740.5	0.04989	833.8	0.082496	874.7	0.118299	869	0.153405	927.3
0.025761	744.6	0.050573	833.2	0.083179	870.9	0.118746	872.5	0.154376	926.3
0.026051	749	0.051151	831.4	0.084071	866.1	0.119325	876.2	0.155112	924.5
0.026341	753.8	0.051597	831	0.084938	865.7	0.120219	881.5	0.155978	922.6
0.026604	757.4	0.052595	830.2	0.085831	864.9	0.120587	885	0.156503	921.7
0.02692	760.5	0.053121	830.1	0.086566	866.2	0.121455	892.6	0.157291	920.5
0.02742	764.3	0.053594	830.1	0.087328	867.5	0.121876	895.3	0.157685	920.2
0.027709	768.3	0.05425	830.8	0.08788	868.2	0.122323	898.7	0.158499	917.9
0.027946	771.9	0.054828	830.7	0.088958	871.6	0.122876	904	0.159103	917.3
0.028262	775	0.055196	831.6	0.089693	873.3	0.123192	906.4	0.159734	917.5
0.028499	777.8	0.055853	832.3	0.09014	873.2	0.123902	913	0.160496	919.3
0.028893	780.8	0.056431	833.4	0.090692	874.7	0.124823	919	0.161021	918.8
0.029235	782.7	0.056852	834.7	0.09148	877.2	0.125664	919.7	0.161941	920.5
0.029577	782.8	0.057351	836.3	0.092085	879.8	0.126189	919.9	0.162545	920.3
0.030023	781.6	0.057693	836.8	0.0929	880.6	0.127003	916.9	0.163438	921.7
0.030443	779.1	0.058611	833.5	0.093425	882.1	0.127895	914.9	0.163911	922.3
0.030968	778.3	0.059294	831.7	0.094108	881.9	0.128762	912.5	0.164568	922.8
0.031651	775.6	0.059819	829.6	0.09466	882.9	0.129497	908.8	0.16533	923.2
0.032123	773.1	0.060213	828	0.095475	885.3	0.130442	906.7	0.16596	923.6
0.032543	771.1	0.061237	825.5	0.095895	885.7	0.131125	905.7	0.166512	923.1
0.033147	769.8	0.062234	823.4	0.096394	887.2	0.131597	905.3	0.167011	922.6
0.033541	769.5	0.062944	825.3	0.096946	888.3	0.132307	905	0.16772	922
0.033987	769.9	0.06368	827.9	0.097577	890.1	0.132727	906.1	0.168245	921.5
0.034461	773.7	0.063995	828.8	0.098312	888.2	0.133515	906.9	0.169033	920
0.034777	776.8	0.064548	833.2	0.098916	885.3	0.133909	906.1	0.169663	918.6
0.035119	780.7	0.064969	836.1	0.099598	882.3	0.134698	909.2	0.170189	918.5
0.035409	785.9	0.065416	840.2	0.100044	878.7	0.135434	910.4	0.170609	917.7
0.035751	792.3	0.06589	845.4	0.100621	875.5	0.135933	910.6	0.17116	917.9
0.036093	796.2	0.066442	850.2	0.101356	870.9	0.136432	912.4	0.172369	919.3
0.036383	799.7	0.066942	855.4	0.10196	867.2	0.137195	915.9	0.173341	918.4
0.036962	806.8	0.067152	856.5	0.102721	864.2	0.137799	916	0.173604	918.5
0.037225	809.9	0.067757	860	0.103482	861.5	0.138351	916.3	0.174418	918.8
0.03762	813.3	0.068335	860.6	0.104165	860.9	0.139402	920.5	0.175048	918.6
0.037909	816	0.06915	864.8	0.104875	864.2	0.139848	919	0.175652	917.6
0.038225	818.6	0.070359	868.9	0.1054	864.5	0.140321	918.9	0.17623	916
0.038488	821.3	0.071279	868.1	0.106005	868.6	0.141215	923.7	0.17686	915.8
0.038883	824.3	0.071725	867.3	0.106689	875.4	0.142108	920.2	0.177754	917.4
0.039303	826.4	0.072277	868	0.107058	877.8	0.142843	919.1	0.178253	918.1
0.03975	827.7	0.073144	868	0.107215	878	0.143394	917.8	0.179015	920.2
0.040276	828.5	0.073538	867.6	0.107847	882.9	0.144051	918	0.179646	922.2
0.040906	829.6	0.073984	868.4	0.108635	882.1	0.144813	918.7	0.180198	923.8
0.041642	830.2	0.074615	868.7	0.10937	880.7	0.145443	918.5	0.18075	925.8
0.04222	830.6	0.074851	869.3	0.109895	878.7	0.146153	918.2	0.181223	927.2
0.042798	831.3	0.075771	871.2	0.110787	876.3	0.146547	917.4	0.181985	929.3
0.04356	831.9	0.076244	872.4	0.11147	872.9	0.14744	917.7	0.182589	928
0.044269	832.9	0.076796	873.1	0.112336	867.9	0.147912	916.9	0.183352	930.1
0.045162	833	0.077532	874.8	0.11307	863.7	0.148649	919.5	0.184402	930.6
0.045793	833.7	0.078215	876.2	0.113701	862.8	0.149358	921.5		
0.046265	832.8	0.079082	877.8	0.114252	860	0.149726	923		
0.046659	833.7	0.079345	878.8	0.115066	858.1	0.149936	923.5		
0.047343	834.4	0.079949	880.2	0.11588	857.7	0.150567	924.8		

Appendix B. Brief Introduction to Python

Python is the name of a general-purpose scripting language designed to be highly readable, as well as an interpreter of that language.[1] When used with add-on modules such as NumPy,[2] SciPy,[3] Pandas,[4] and Matplotlib,[5] it can become a powerful tool for scientific computing and data analysis. These add-on packages are readily available through the Anaconda Python Distribution,[6] the Python Package Index (PyPI),[7] and they may also be available through the package manager of a Linux distribution. There are two main versions of the Python language, the legacy Python 2.x, which will cease being supported in 2020,[8] and Python 3.x. This introduction is meant to pertain to the latter, though most of what is discussed here applies to both versions.

B.1 Basic Syntax

The basic arithmetic operators used in Python are typical of most scripting and programming languages, so "+" and "−" are of course used for addition and subtraction, and "⋆" and "/" are used for multiplication and division. The exponentiation operator is "**". The default precedence of these arithmetic operations also follows the conventions typical in both mathematical notation and other programming languages, that is, exponentiation is done before multiplication and division, which are in turn done before addition and subtraction. Parentheses are used for grouping operations together. Arithmetic operators are also used in some non-arithmetic contexts. For example, the "+" operator is also used to concatenate character strings, and the "⋆" is used for repeating sequences, e.g., `"a"*4` is `"aaaa"`, where both `"a"` and `"aaaa"` are strings.

Identifiers in Python, such as variables and function names, consist of a combination

[1] Python Software Foundation. Welcome to Python.org. c2018 [accessed 2018 May]. https://www.python.org/.

[2] NumPy developers. NumPy. c2018 [accessed 2018 Apr]. http://www.numpy.org/.

[3] SciPy developers. SciPy. c2018 [accessed 2018 May]. http://www.scipy.org/.

[4] Pandas developers. pandas: powerful Python data analysis toolkit. c2017 [accessed 2018 May]. http://pandas.pydata.org/pandas-docs/stable/index.html.

[5] Matplotlib development team. Matplotlib: Python plotting—Matplotlib 2.2.2 documentation. c2018 [accessed 2018 May]. https://matplotlib.org/.

[6] Anaconda, Inc. Anaconda. c2018 [accessed 2018 Mar]. https://anaconda.com.

[7] Python Software Foundation. PyPI—the Python Package Index. c2018 [accessed 2018 May]. https://pypi.org/.

[8] Benjamin Peterson. PEP 373 — Python 2.7 Release Schedule. c2008 [accessed 2018 Jul]. https://www.python.org/dev/peps/pep-0373/#update

of letters, numbers, and underscores ("_"). Identifiers, though, cannot start with a number, nor can they be certain keywords in Python, such as `if`, `while`, and so on. While an identifier can start with an underscore, this is generally only done when the identifier is used as some internal detail, such as functions or variables meant only to be used within a Python module.

Numbers in Python are represented mostly straightforwardly, such that, for example, `2.3` represents the real number 2.3, and `1` represents the integer 1. Scientific notation is represented such that `1.23456e7` means 1.23456×10^7. Complex numbers can be represented as well; the complex number $1.3 + 2.1i$ is expressed as `1.3 + 2.1j`. (The imaginary unit i is represented as `1j` in Python, not as `j`. The latter is just a variable name.) Internally, a Python interpreter represents real numbers approximately as so-called floating-point numbers,[9] much as interpreters and compilers of other languages do. This representation is distinct from a Python interpreter's internal representation of integer values. Among other things, this means that the expressions `1` and `1.0` do not mean exactly the same thing in Python. The former indicates an integer value, while the latter represents a floating-point number.

Other literal quantities in Python are character strings and Boolean values. Character strings are delimited by either single or double quotes. For example, either `'a string'` or `"a string"` represents a string consisting of the characters "a", a space, "s", "t", "r", "i", "n", and "g". Multiline strings may be delimited by a triplet of single or double quotes, that is, either "`'''`" or "`"""`". Certain sequences in strings that begin with a backslash ("\") are interpreted specially. In particular, `"\n"` indicates a newline, and `"\\"` indicates a literal backslash. Boolean values are represented in Python by the keywords `True` and `False`.

Python has a few other special characters and character sequences. One of these is the character "#", which indicates the start of comment text. Everything from this character to the end of the line is ignored by the Python interpreter. Another is the operator "=", which is used to assign values to variables. The following are examples of assignment statements, with comments indicating what the assignment statement is doing:

```
x = 1          # Assigning 1 to the variable x
```

[9] Goldberg D. What every computer scientist should know about floating-point arithmetic. ACM Comput Surv. 1991;23(1):5–48.

```
y = 2e5         # Assigning 200000.0 to the variable y
z = 6.3e-2      # Assigning 0.063 to the variable z
```

In general, statements in Python are terminated at the end of a line, *provided that they are complete statements*. For example,

```
a = x + y + z
```

is a complete statement on a single line that adds the variables `x`, `y`, and `z` together and assigns the resulting sum to `a`, while

```
a = x + y +
```

is not complete and is treated as an error in Python. There are a couple solutions to this. The first is to end the incomplete statement with a backslash. This indicates to the Python interpreter that it should look to the next line to try to complete the statement. Accordingly, the following Python code is correct:

```
a = x + y + \
  z
```

However, if an incomplete statement includes a opening parenthesis, bracket, or brace (i.e., "`(`", "`[`", or "`{`"), then Python attempts to complete the statement by whatever is between the end of the current line and the corresponding closing parenthesis, bracket, or brace (i.e., "`)`", "`]`", or "`}`"). Accordingly, the following is essentially equivalent to the previous example Python code:

```
a = (x + y +
  z)
```

In this code, brackets or braces could not have been used because they have special meanings in Python that are illustrated in Sections B.4 and B.5.

Assignment and arithmetic can be combined in Python. For example, in the following code,

```
x += 2
```

the variable `x` is incremented by 2. If "`-=`" were used instead, `x` would be decremented by 2, and "`*=`" and "`/=`" would multiply and divide `x` by 2, respectively.

Relational and logical operations are usually used with the control structures discussed in Section B.11. The relational operators "<", ">", "<=", ">=", "==", and

"`!=`" mean what they do in most programming languages, so for example, the expression "`x < y`" tests if `x` is less than `y`, "`x >= y`" tests if `x` is greater than or equal to `y`, "`x == y`" tests if `x` equals `y`, and "`x != y`" tests if `x` is not equal to `y`. Negation is indicated by the keyword "`not`", so that `not True` is `False` and vice versa. Similarly, Boolean operations "and" and "or" are represented by keywords as well, namely, `and` and `or`. The expression `x and y` is `True` only if both `x` and `y` are `True`, and the expression `x or y` is `False` only if both `x` and `y` are `False`. Both these operators short-circuit; if the first operand of the `and` operator is `False`, the second operand is never evaluated, and if the first operand of the `or` operator is `True`, the second operand is never evaluated.

B.2 Methods and Attributes

Like many languages, Python has functions, which take one or more arguments, perform some sequence of operations with those arguments, and possibly return values. For example, `abs(x)` is the absolute value of `x`. These could be considered free functions, in the sense that they exist apart from any particular object. However, there are also functions that are essentially part of an object and are invoked in a special way that accounts for this. These may be described as member functions but are more commonly called *methods*. Whereas a free function `my_func` may be invoked as `my_func(x, y, z)`, a method `my_method` bound to an object `x` would be invoked as `x.my_method(y,z)`. Different kinds of objects have different methods associated with them.

For example, strings have several methods associated with them. The method `startswith` returns a Boolean value that indicates if a string starts with a certain string of characters. Example usages are shown:

```
f = "barbaz"

bar_start = f.startswith("bar") # bar_start is True
f_start = f.startswith("f")     # f_start is False
```

Methods do not have to be applied to variables; for example, the method invocation `"barbaz".startswith("bar")` is perfectly valid.

As another example, the method `conjugate` is associated with numbers and returns the complex conjugate of a number. For example, `(1 + 2j).conjugate()` returns `1 - 2j`.

Objects can have attributes as well as methods. Loosely speaking, attributes may be described as methods that not only take no arguments (like the `conjugate` method mentioned previously) but also do not need parentheses to invoke them. For example, if `z` is `3 + 5j`, then `z.real` (not `z.real()`!) is the real part of `z`, or 3, and `z.imag` (not `z.imag()`!) is the imaginary part of `z`, or 5.

B.3 Format Strings

Format strings are a kind of template. They contain pairs of braces (i.e., "`{}`") that can be substituted by other expressions. An example format string would be `"Re(z) = {}, Im(z) = {}"`. To use a format string, one uses the `format` method, as shown in the following example code:

```
z = 2 + 3.2j
out_string = "Re(z) = {}, Im(z) = {}".format(z.real, z.imag)

print(out_string)
```

This prints out the string "`Re(z) = 2.0, Im(z) = 3.2`." Doubled braces in a format string, "`{{`" or "`}}`", are replaced by "`{`" or "`}`", respectively, by the `format` method.

Typically, what a format string substitutes for "`{}`" is the string representation of an object returned by the function `str`. In this case, `str(z.real)` is `"2.0"`, and `str(z.imag)` is `"3.2"`. Of course, the string representation of a string is the string itself. Objects other than numbers and strings have string representations as well, though the usefulness of these representations may vary.

B.4 Lists and Tuples

Both lists and tuples are sequences of arbitrary objects. For example, a list may be created and assigned to `list_x` as follows:

```
list_x = [42, "tuna", 1 + 3j, 2.0, 7]
```

Similarly, a tuple may be created and assigned to `tuple_x` as follows:

```
tuple_x = (42, "tuna", 1 + 3j, 2.0, 7)
```

Accessing the elements of tuples and lists also works the same way. Both use 0-based indexing, so the first element of a list or tuple `x` is `x[0]`, the second element is `x[1]`, and so on. So-called slices can also be used to access subsets of a list or tu-

ple. For example, `list_x[1:4]` and `tuple_x[1:4]` return the second through the fourth elements of `list_x` and `tuple_x`, respectively, that is, `"tuna"`, `1 + 3j`, and `2.0`. The slice `1:4` starts with the second element because the number before the colon in the slice, 1, is the index of the second element (due to 0-based indexing). The slice ends with the fourth element because the number after the colon in the slice, 4, is the index of the fourth element plus 1 (i.e., 3 + 1).

Another similarity between lists and tuples is that the number of elements in both is determined with the same function, `len`. The number of elements in a list or tuple `x` is `len(x)`.

There is a significant difference between lists and tuples, though; only the former is mutable. For example, one may change the first element of `list_x` to 54 with the statement `list_x[0] = 54`. However, attempting to execute the statement `tuple_x[0] = 54` leads to the error message, `TypeError: 'tuple' object does not support item assignment`. One can also append to a list (but not a tuple). For example, the following Python code appends two items to a list `y`, which happens to initially be empty:

```
y = []
y.append("item 1")
y.append("item 2")
```

Once this code is executed, `y` becomes the list `["item 1", "item 2"]`.

Both lists and tuples can be unpacked. For example, if `x` is the list `["one", "two"]` or the tuple `("one", "two")`, then the assignment statement

```
x1, x2 = x
```

leads to the variables `x1` and `x2` being set to the strings `"one"` and `"two"`, respectively.

B.5 Dictionaries

In Python, a dictionary is a set of key/value pairs. The following Python code shows how a dictionary may be constructed:

```
month_str_to_num = {
   "jan": 1, "feb":  2, "mar":  3, "apr":  4,
   "may": 5, "jun":  6, "jul":  7, "aug":  8,
   "sep": 9, "oct": 10, "nov": 11, "dec": 12
```

```
}
```

In this example, the keys are strings that stand for calendar months, and the value associated with each string is the number corresponding to that month. Keys do not have to be strings. Numbers and tuples of numbers can be keys as well. Not all objects are suitable keys, though. For example, lists cannot be keys. However, the values in a dictionary can be any sort of object.

Accessing elements of a dictionary is somewhat similar to accessing elements of a list or tuple. For example, in the previous example dictionary, `month_str_to_num["jun"]` is 6. The number of key-value pairs in a dictionary is determined via the same function `len` used to obtain the number of elements in a list or tuple, so `len(month_str_to_num)` is 12.

Additional key/value pairs may be added to a dictionary after its initial creation. For example, in the following code,

```
my_data = {}
my_data["var1"] = [1,2,3]
my_data["var2"] = [(3,2), (46,7), (2,5)]
```

initially `my_data` is an empty dictionary, and the dictionary entries `my_data["var1"]` and `my_data["var2"]` are created when values are assigned to them. One can also use the `update` method to add or modify a dictionary. For example, the following code replaces the value associated with the key `"var1"` with the number 55, and adds a new key/value pair, `"var3"/"new value"`:

```
my_data.update({"var1": 55, "var3": "new value"})
```

The keys of a dictionary can be accessed via the `keys` method. For example, `list(my_data.keys())` returns the list `["var1", "var2", "var3"]`. Similarly, the key/value pairs in a dictionary can be accessed by the `items` method, with `list(my_data.items())` returning the list of tuples `[("var1", 55), ("var2", [(3, 2), (46, 7), (2, 5)]), ("var3", "new value")]`. (The methods `keys` and `items` technically return so-called `iterators`, which are discussed more in Section B.11. The function `list` converts the iterator to a list.) To determine if a key is already in a dictionary, one can use the keyword operator `in`. For example, `"var1" in my_data` returns `True`, whereas `"var4" in my_data` returns `False`.

B.6 Function Definition and Invocation

Python has several built-in functions, and users can define their own functions as well. A somewhat contrived example of a function definition is shown:

```
from cmath import sqrt

def qd_formula(a, b = 0, c = 0):
   "Solves the quadratic equation a*x**2 + b*x + c = 0"

   sqrt_b2_minus_4ac = sqrt(b**2 - 4*a*c)
   two_a = 2*a

   return ((-b - sqrt_b2_minus_4ac)/two_a,
           (-b + sqrt_b2_minus_4ac)/two_a)
```

All of the indented lines after the colon ("`:`") on the line starting with `def` form the body of a function named `qd_formula`, which implements the quadratic formula $x = (-b \pm \sqrt{b^2 - 4ac})/2a$. The very first line in the body is a string that documents what the function does. The `sqrt` function implements the square root. It has been imported from the `cmath` module, which contains definitions of functions that work with complex numbers, so `sqrt(-1)` returns `1j` instead of producing an error. (More on importing modules is discussed in Section B.7.) Finally, the function returns a tuple containing the two solutions of the quadratic formula.

This function can be invoked several different ways. For example, it could be invoked simply as `qd_formula(1,2,2)`, which returns the complex values $-1 \pm i$. It could also be invoked as `qd_formula(a = 1, b = 2, c = 2)` or even `qd_formula(b = 2, c = 2, a = 1)`, and the same result would be obtained. These latter ways of invoking the function involve so-called *keyword arguments*. One can even invoke the function as `qd_formula(**my_args)`, where `my_args` is the dictionary `{"a": 1, "b": 2, "c": 2}`. This function also has default values for arguments `b` and `c`, so that arguments that are not explicitly passed are assigned these values. For example, `qd_formula(1,2)` means the same thing as `qd_formula(1,2,0)`, and `qd_formula(1, c = 2)` means the same thing as `qd_formula(1,0, c = 2)` or `qd_formula(1,0,2)`. While this example is contrived, the use of keyword and default arguments is not. Many Python functions (such as those from the Matplotlib and Pandas modules) have a large number of arguments, and to make that large number of arguments manageable, most of these argument have default values. The few arguments that

need to be supplied explicitly are then usually supplied by keyword for the sake of readability.

Small functions can also be defined via the `lambda` keyword. For example, in the following contrived example,

```
add2 = lambda a, b: a + b
```

a function `add2` is defined that adds two numbers. It works like an ordinary function, so `add2(1,2)` returns 3. Only a single expression is allowed after the colon in a `lambda` expression. Also, in practice, a `lambda` expression, such as the one on the right-hand side of the "=" operator in the above Python code, is usually not assigned to a variable. Instead, it is often used as an argument to another function, especially in cases where a simple one-off function are needed.

B.7 Importing Modules

Strictly speaking, very few functions are directly built-in to Python. Rather, most functions belong to modules, either those that are part of Python's standard library or third-party modules such as NumPy. In Section B.6, there is a brief example where a particular function `sqrt` is imported from the module `cmath`. Alternatively, the whole module could have been imported with the following statement:

```
import cmath
```

If this were used instead of "`from cmath import sqrt`" in Section B.6, then the definition of `qd_formula` would have had to use `cmath.sqrt` in place of `sqrt`. In general, when a module is imported, in order to use the functions within it, one must prefix their names with the name of the module and a period ("`.`").

Modules may also contain submodules. For example, the `os` module of Python's standard library contains a submodule named `path`, which contains various functions for manipulating file names. Accordingly, when the submodule is imported with the following Python code,

```
import os.path
```

those functions have the "`os.path.`" prefix.

To reduce typing, modules can be given a short name via the `as` keyword of an `import` statement. For example, it is common to import the NumPy module as

follows:

```
import numpy as np
```

Accordingly, functions from the NumPy module are then prefixed with "`np.`" instead of using the "`numpy.`" prefix.

A module can be created by simply writing a Python file containing function definitions. For example, one may write a file named `my_functions.py` with the following contents:

```
from cmath import sqrt

def qd_formula(a, b = 0, c = 0):
   sqrt_b2_minus_4ac = sqrt(b**2 - 4*a*c)
   two_a = 2*a

   return ((-b - sqrt_b2_minus_4ac)/two_a,
           (-b + sqrt_b2_minus_4ac)/two_a)

def add2(a,b):
   return a + b
```

If a Python script is executed from the same directory as the file `my_functions.py`, then it can import the module by having the following statement in it:

```
import my_functions
```

Then, the script can invoke the functions in the module by prefixing them with "`my_functions.`":

```
x1, x2 = my_functions.qd_formula(1,2,3)
y = my_functions.add2(x1, x2)
```

There are other ways to create modules, as well as ways of installing them so that their contents do not need to be in the same working directory as a Python script, but these are outside the scope of this introduction.

B.8 Cross-platform File Path Functions

Python runs on several platforms, including Windows, MacOS, and Linux. In general, various platforms have different ways of specifying paths to files. For example, on Windows, typically a path to a file is specified as `path\to\file`, while on MacOS and Linux, it is specified as `path/to/file`. However, to ensure portability, it is best to specify the file path in Python as `os.path.join("path",`

`"to", "file")`. This constructs a path with backslashes on Windows and forward slashes on MacOS and Linux.

To specify a parent directory in a cross-platform manner, one should use the variable `os.pardir`. For example, if a script is in the directory `path/to/script_dir` (or `path\to\script_dir`), and it needs to access the file `path/to/my_data` (or `path\to\my_data`), then the file `my_data` is in the parent directory of `script_dir`, and the script in `script_dir` can thus specify the file as follows:

```
my_data_file_name = os.path.join(os.pardir, "my_data")
```

Of course, to use both `os.path.join` and `os.pardir`, the module `os` must be imported.

B.9 Third-Party Data Structures

These data structures come from modules that are add-ons to Python, rather than a part of its standard library.

B.9.1 NumPy Arrays

In general, a NumPy array contains an n-dimensional grid of values, where the values in an array must all have the same type (e.g., integer, floating-point, complex). If $n = 2$, then the array resembles a matrix. If $n = 1$, then the grid of values reduces to a sequence of values, like a row or column vector. The following code creates a 2-D array that represents a 2×3 grid of floating-point values:

```
import numpy as np

A = np.asarray([[1.0, 2.0, 3.0],
                [4.0, 5.0, 6.0]])
```

The function `asarray` can be used to construct an array. Since a 2-D array is being constructed, the argument to `asarray` is a list of lists, where each element of the list represents a row in the resulting array. To create a 1-D array, one simply passes a list of values to `asarray`:

```
v = np.asarray([3.5, 7.8, 6.2, 1.5])
```

(There is also a function `array` in NumPy that can be used to construct arrays. The advantage of using `asarray` instead of `array` is that if `asarray` is given a NumPy array as an argument, it simply returns that argument without making a

copy of it, thus saving both time and memory.)

Accessing elements of a 1-D array is much like accessing elements of a list or tuple. Indexing is still 0-based. The slice syntax may be used as well, so for example, given the definition of `v` in the previous Python code, `v[1:3]` is the 1-D array with elements 7.8 and 6.2. Accessing elements of an array is similar to accessing elements of a vector. For example, given the previous definition of `A`, `A[0,1]` is the element in its first row and second column (i.e., 2). The expression `A[1,1:3]` indicates the second and third elements of the second row (i.e., 5 and 6). Also, `A[1,:]` is the whole second row (i.e., 4, 5, and 6), and `A[:,2]` is the third column (i.e., 3 and 6).

Arithmetic operators work elementwise on arrays. For example, if `A1` and `A2` both have dimensions $n_1 \times n_2 \times n_3$, then `A1*A2` is such that its elements are `A1[0,0,0]*A2[0,0,0]`, `A1[0,0,1]*A2[0,0,1]`, and so on. Arithmetic operations with scalars and arrays are defined as well. If `b` is a scalar, then `A1**b` is such that its elements are `A1[0,0,0]**b`, `A1[0,0,1]**b`, and so on. There are also functions in NumPy that operate elementwise on arrays. For example, `np.sqrt(A1)` is such that its elements are $\sqrt{\texttt{A1}[0,0,0]}$, $\sqrt{\texttt{A1}[0,0,1]}$, and so on. These functions also can also operate on lists and tuples that can be converted to arrays; `np.sqrt([1.0, 2.0])` is essentially equivalent to `np.sqrt(np.asarray([1.0, 2.0]))`.

NumPy arrays have various attributes. For example, the `T` attribute is the transpose of an array, so `A.T` (where `A` is still defined as shown above) is the following 3×2 array:

$$\begin{matrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{matrix}$$

Another attribute is `shape`, which is a tuple containing the dimensions of the array. For example, `A.shape` is `(2, 3)`.

B.9.2 Pandas Data Frames

The data frames defined by the Pandas module are table-like data structures. The data in a given column of a data frame must be of the same type, but different columns may have different types of data. Often, data frames are created by reading in external data. For example, given a CSV file named `my_data.csv` with the

following contents,

```
AA,BB,Test
1.8, 2, 44.5
3.1, 2, 32.1
0.5, 1, 55.3
0.4, 6, 66.3
```

a data frame named `df` may be created as follows.

```
import pandas

df = pandas.read_table("my_data.csv", sep = ",")
```

Here, the argument "`sep = ","`" indicates that a comma should be taken as the separator between two elements of a row. Equivalently, the data frame may be read in the following way:

```
df = pandas.read_csv("my_data.csv")
```

Columns of data frames can be accessed via the name of the column. For example, `df["AA"]` is a sequence with the components 1.8, 3.1, 0.5, and 0.4. (This sequence is actually another data type from Pandas called a series, which behaves much like a 1-D NumPy array.) Data frames also have two attributes, `iloc` and `loc`, which can be used to access the elements of a dataframe. The attribute `iloc` allows the elements to be accessed like elements of a 2-D array, so for example `df.iloc[0,2]` returns the third element of the first row, or 44.5, and `df.iloc[1:3,1]` accesses the second and third rows of the second column, or the sequence of elements 2 and 1. The attribute `loc` accesses elements by the labels of a data frame. For example, `df.loc[:,"AA"]` returns the column with the name `"AA"`, while `df.loc[1:2,"BB"]` returns the second and third rows of column `"BB"`.

The `shape` attribute of a data frame is a tuple whose first element is the number of rows in the data frame and whose second element is the number of columns in it. For example, `df.shape` is `(4,3)`.

B.10 Plotting with Matplotlib

The capabilities and limitations of the plotting functionality in Matplotlib may be shown with some simple examples. Suppose, for instance, that one wishes to plot the following NumPy arrays (where the NumPy module has been imported as `np`):

```
# x is an array of 10 evenly spaced values from 1.0 to 5.0
x = np.linspace(start = 1.0, stop = 5.0, num = 10)
x_sq = x**2
```

This may be done simply with the following Python code:

```
import matplotlib.pyplot as plt
import os.path

plt.figure(figsize = (2.0,2.0))

plt.plot(x, x_sq)

plt.xlabel("x")
plt.ylabel("$x^2$")
plt.tight_layout()

plt.savefig(os.path.join("plot_files", "plot_example1.pdf"))
```

The function `figure` from the `matplotlib.pyplot` submodule is used here to set the width and height of the figure to 2 inches. The plotting is done, of course, with the `plot` function, which by default plots the curve shown in Fig. B-1a. The functions `xlabel` and `ylabel` are used to set the labels of the x- and y-axes, respectively. The argument of the latter is bracketed by dollar signs to indicate that it should be treated as LaTeX,[10] which causes the character "`^`" to be treated as an indicator that "`2`" is a superscript. It also causes the "`x`" in the label to be shown in italics. Because the figure size is small, the `tight_layout` function is needed so that the axis labels appear in the plot. Finally, `savefig` is used to save the plot to a file named `plot_example1.pdf` in the directory `plot_files` (assuming that the directory already exists).

Suppose one wishes to plot the following variable as well:

```
two_x_sq = 2*x_sq
```

One might first attempt to plot the above variable and the previous ones on the same graph as follows:

```
plt.figure(figsize = (2.0,2.0))

plt.plot(x, x_sq)
plt.plot(x, two_x_sq)
```

[10] LaTeX project team. LaTeX: a document preparation system. c2018 [accessed 2018 May]. `https://www.latex-project.org/`.

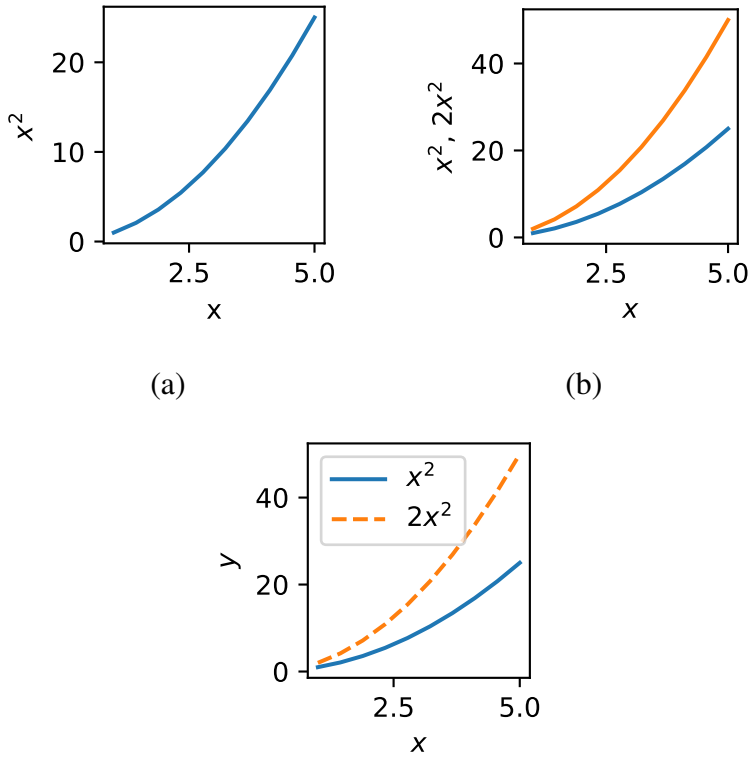


Fig. B-1 Example plots used to illustrate the plotting features of the Python module Matplotlib

```
plt.xlabel("$x$")
plt.ylabel("$x^2$, $2x^2$")
plt.tight_layout()

plt.savefig(os.path.join("plot_files", "plot_example2.pdf"))
```

Here, the `plot` function is used twice, first to plot x^2 versus x and then to plot $2x^2$ versus x. Also, the label for the x-axis is bracketed in dollar signs so that it is presented in italics, just as the label for the y-axis is.

The resulting plot, shown in Fig. B-1b, has a couple problems. First, it is not clear which curve belongs to which variable. Second, the two curves are only distinguished by different line colors. For plots that may be later printed in black and white (or for the color-blind), this can be a problem. To fix the first problem, a legend is added. To fix the second problem, line styles are explicitly supplied:

```
plt.figure(figsize = (2.0,2.0))

plt.plot(x, x_sq, label = "$x^2$", linestyle = "solid")
plt.plot(x, two_x_sq, label = "$2x^2$", linestyle = "dashed")

plt.xlabel("$x$")
plt.ylabel("$y$")
plt.legend(loc = "upper left")
plt.tight_layout()

plt.savefig(os.path.join("plot_files", "plot_example3.pdf"))
```

The results of this code are shown in Fig. B-1c.

B.11 Control Structures

Control structures allow for more complicated logic to be used in Python scripts. A few of these structures are shown in this section.

B.11.1 Branching: `if`, `elif`, `else`

The `if`/`elif`/`else` structure is as follows:

```
if condition:
   # Statements executed if condition is True
elif other_condition:
   # Statements executed if other_condition is True
else:
   # Statements executed if none of the conditions are true
```

Here, the comments substitute for statements that would be used in an `if`/`elif`/

`else` structure in practice. These comments are indented in same way that the actual statements would need to be indented, since the Python interpreter uses the indentation itself to determine which statements belong to the `if`, `elif`, and `else` statement blocks. The variables `condition` and `other_condition` stand in for expressions that may evaluate to either `True` or `False`, such as, for example, `x <= x_threshold`. Both the `elif` and `else` blocks are optional. The keyword `elif` is short for "else if". Without it, the previous Python code would be written as follows:

```
if condition:
   # Statements executed if condition is True
else:
   if other_condition:
      # Statements executed if other_condition is True
else:
   # Statements executed if none of the conditions are true
```

B.11.2 Iteration: `while` and `for`

The `while` loop is as follows:

```
while condition:
   # Statements executed so long as
   # condition is True
```

Here, the comments substitute for statements that would be used in a practical `while` loop, and the variable `condition` stands in for an expression that may evaluate to either `True` or `False`, such as `err > threshold`. Indented statements below the `while` clause are iterated (i.e., repeatedly executed) until `condition` becomes `False`, so these statements should include some statement that would alter `condition`, or else the `while` loop iterates forever (or more realistically, until someone kills or interrupts the Python session). A trivial example of a terminating `while` loop (i.e., one that stops iterating eventually) is as follows:

```
x = 10
while x > 0:
   x -= 1
```

Since the body of the `while` loop keeps decrementing `x` by 1, the loop condition `x > 0` eventually becomes false. One may also force a `while` loop to stop iterating after a fixed number of iterations as follows:

```
threshold = 1e-6
err = 2*threshold
```

```
i = 0
while err > threshold:
   estimate_and_err = do_estimate(A,B,C)
   err = estimate_and_err[1]

   i += 1
   if i > 1000:
      break
```

The `break` statement causes the loop to terminate once `i` exceeds 1000, even if the condition `err > threshold` has not yet been reached. This might be done, for example, in case the (made up) function `do_estimate` does not successfully reduce `err` as much as it should.

Alternatively, the logic of the previous `while` loop can be rewritten with a `for` loop:

```
threshold = 1e-6

for i in range(1000):
   estimate_and_err = do_estimate(A,B,C)
   err = estimate_and_err[1]

   if err <= threshold:
      break
```

This loop is intended to execute the indented statements below the `for` clause for a fixed number of iterations. Provided that `err` always exceeds `threshold`, this particular loop iterates 1000 times. Of course, here the `break` statement can cause the `for` loop to terminate before 1000 iterations have completed, provided that the condition `err <= threshold` is reached. Also, at each iteration of the loop, the loop variable `i` takes on a different value, 0 for the first iteration, 1 for the second, and so on, until `i` reaches the value of 999. Since the value of `i` is not used in the body of this particular loop, the change in its value does not appear to matter, but in a loop such as this:

```
for i in range(len(v1)):
   v2[i] = do_something(v1[i])
```

where the loop variable `i` is used to access successive values of `v1` and `v2`, which may be lists, tuples, or 1-D arrays.

The expression `range(N)` used in the previous `for` loops expresses the sequence

0, 1, 2, . . . , N − 1. The sequence generated by the `range` function does not have to start at zero. The expression `range(N_start, N)` expresses the sequence `N_start`, 1, 2, . . . , N − 1. To be more precise, the `range` function returns an iterator, a generator that returns the next element of a sequence with each iteration of a `for` loop. The iterator returned by `range` generates a monotonic, evenly spaced sequence of integers without holding all the values of the sequence in memory at once.

A `for` loop does not necessarily have to involve the `range` function. Rather, one may iterate over any sequence or iterator, including lists, tuples, dictionaries, and NumPy arrays. For example, the following Python code,

```
list_abD = ["a", "b", "D"]
for l_abD in list_abD:
   print(l_abD)
```

simply prints out the elements of `list_abD`, that is, `"a"`, `"b"`, and `"D"`, and the following code,

```
dict_abD = {"a": 1, "b": 2, "D": 3}
for k, v in dict_abD.items():
   print("Key = {}; Value = {}".format(k,v))
```

prints the key/value pairs of `dict_abD`. This example also shows the use of sequence unpacking in `for` loops. Essentially, it is equivalent to the following code,

```
dict_abD = {"a": 1, "b": 2, "D": 3}
for key_val_tuple in dict_abD.items():
   k, v = key_val_tuple
   print("Key = {}; Value = {}".format(k,v))
```

except that the intermediate variable `key_val_tuple` is not used.

One can also use the iterator `enumerate` to generate an index that accompanies successive elements of a list, as shown in the following code:

```
list_abC = ["a", "b", "C"]
for i, l_abC in enumerate(list_abC):
   print("Item {}: {}".format(i, l_abC))
```

This prints `Item 0: a`, `Item 1: b`, and `Item 2: C`.

B.11.3 Abbreviated `for` Loops: List and Dictionary Comprehensions

Certain kinds of `for` loops can be presented in abbreviated form. For example, the `for` loop in the following Python code builds a new list, `list_of_lengths`, where each element is the result of applying the function `len` to the corresponding elements of a preexisting list, `list_of_lists`:

```
list_of_lists = [
   [3,4,2,5],
   [6,7,1],
   [5],
   [3,5],
   [1,2,3,4,5,6],
   [2]
]

list_of_lengths = []
for l in list_of_lists:
   list_of_lengths.append(len(l))
```

This `for` loop can be expressed more concisely with what in Python is called a *list comprehension*:

```
list_of_lengths = [len(l) for l in list_of_lists]
```

Here, `list_of_lengths` is `[4, 3, 1, 2, 6, 1]`. List comprehensions can have a filter, as in the following example:

```
list_of_len_gt_1 = [len(l) for l in list_of_lists if len(l) > 1]
```

Here `list_of_len_gt_1` is `[4, 3, 2, 6]`, which is like `list_of_lengths` except that all entries not satisfying condition `len(l) > 1` are removed.

Dictionary comprehensions are much like list comprehensions. For example, the `for` loop in the following Python code that generates a dictionary,

```
param_names = ["A", "B", "n", "C", "m"]

param_to_index = {}
for i, param in enumerate(param_names):
   param_to_index[param] = i
```

can be expressed as follows:

```
param_to_index = {param: i for i, param in enumerate(param_names)}
```

Both the explicit `for` loop and the dictionary comprehension create dictionaries

that map a string (indicating a parameter name) to a numerical index.

B.11.4 Exception Handling: `try` and `except`

When the Python interpreter encounters an error, it often raises what is called an *exception*, an object that contains information about an error, such as a message on what the error is and the line number in the Python code that raises the exception. Raising exceptions typically halts the execution of a Python script, which may not be desirable. To handle the exceptions that may be raised, one can use `try` and `except` statements. A simple example example usage is as follows:

```
try:
   # do_something_dodgy returns True if it works. It may return
   # False if it doesn't, or it may just go awry.
   dodgy_flag = do_something_dodgy(A,B,C)
except:
   print("WARNING: do_something_dodgy failed!")
   dodgy_flag = False
```

Again, Python uses indentation to determine which statements are part of the `try` block and which are part of the `except` block.

In the previous example, all possible exceptions are handled. However, in many cases, one should handle very specific exceptions, such as in the following example function definition:

```
def str_to_num(x):
   "A non-robust converter of strings to real numbers"

   try:
      return int(x)
   except ValueError:
      return float(x)
```

This function first attempts to convert a string to an integer via the `int` function. If this function raises a `ValueError` exception, indicating that it has been passed a string argument that cannot be interpreted as an integer (e.g., `"1.0"`), then an attempt is made to convert the string to a floating-point number via the `float` function.

B.11.5 Context Management: The `with` Statement

The `with` statement has two forms, either

```
with my_expr:
```

```
   # Statements in "with" block
```

or

```
with my_expr as my_var:
   # Statements in "with" block
```

In the second form of the `with` statement, the Python expression `my_expr` is an object that is assigned to the variable `my_var`. The statements in the `with` block, which are the statements indented below the `with` clause, are executed in a context that depends on the type of `my_expr`. To be more precise, the object `my_expr` has two methods, `__enter__` and `__exit__`, the first of which is invoked at the start of the `with` block, and the second of which is invoked either after the end of the `with` block or if an unhandled exception is encountered.

A common usage of the `with` statement is in opening files, such as in the following example:

```
with open("out_file.txt", "w") as out_file:
   out_file.write("Stuff and ")
   out_file.write("More Stuff\n")
```

where the expression `open("out_file.txt", "w")` opens the file named `out_file.txt` so that it can be written. The object representing this open file is assigned to the variable `out_file`, which is used to write the string `"Stuff and More Stuff"` to the file. Afterward, the file is closed. If the statements in the `with` block somehow raise an exception, the file is still closed.

B.12 Saving Python Objects to Pickle Files

An object in Python, such as one of the data structures described in Sections B.4, B.5, and B.9, can often be *pickled*, that is, serialized as a stream of bytes, which can then be saved to what here is called a pickle file. This file can be used to read in the object into another Python session. The following code shows an example of saving an object:

```
import pickle

xyz = {"x": (3,4,2), "y": 3 + 2j, "z": "string_val"}

with open("xyz.pkl", "wb") as f:
   pickle.dump(xyz, f)
```

The function `dump` from the `pickle` module (which is part of Python's standard library) writes the dictionary `xyz` to a file object `f` associated with the file named `xyz.pkl`. The `"wb"` in the second argument to `open` is important; it means that the file is opened for writing in binary mode, so that the contents of it are treated as a stream of bytes rather than as text. The `dump` function does not work otherwise.

In another Python session, the dictionary can be read in with the `load` function from the `pickle` module and printed as follows:

```
import pickle

with open("xyz.pkl", "rb") as f:
   xyz_take_2 = pickle.load(f)

print(xyz_take_2["x"])
print(xyz_take_2["y"])
print(xyz_take_2["z"])
```

As can be seen from this example, when an object is read from a file and stored in a variable, this variable need not have the same name as the variable that stored the object in a previous session. Not all objects can be pickled. However, attempting to pickle an unpicklable object raises a `PicklingError` exception.

The format of the byte stream to which pickled objects are serialized is controlled by the optional argument, `protocol`, of the `dump` function. At the time of writing, by default, the value of this optional argument is 3, indicating that the format used is one that can be read by Python 3.x interpreters but not Python 2.x interpreters. Setting `protocol` to a lower value uses an earlier, more backward-compatible format. Since the byte stream format is stable and not supposed to have undocumented changes with new Python versions, it is possible, if not recommended, to use pickle files for long-term storage.

Appendix C. Python Code for Bayesian Analysis

These are the contents of Python module files that have been used for Bayesian analyses of strength models. Comments of the form `#!{...}` can be ignored, since they are meant to be read by tools that extract source code fragments. Documentation of the parameters and return values of functions follows the guidelines of the Numpydoc docstring guide.[1]

C.1 Module File `bayes_stress_strain_utils.py`

```
import numpy as np
import scipy
import pandas
import matplotlib.pyplot as plt
import itertools
import json
import pickle
import gzip
import os
import time

import warnings

try:
    import pystan
    _pystan_available = True
except:
    _pystan_available = False

# If PyStan is not installed, then there is no point in loading the
# stan_utility module (and it will not be importable anyway). This
# should not be an issue for those who do not plan to use PyStan.
if _pystan_available:
    import stan_utility

try:
    import pymc3 as pm
    _pymc3_available = True
except:
    _pymc3_available = False

# Function internal to module
def _ensure_path_to_file_exists(file_name):
    """Ensures that path to a file exists, and if not creates it.

    Parameters
    ----------

    file_name : str
        File name
```

[1] Numpydoc maintainers. Numpydoc docstring guide. c2017 [accessed 2018 May]. `https://numpydoc.readthedocs.io/en/latest/format.html`

```
    """

    # Obtaining what will be the absolute path of the file
    abs_path_name = os.path.abspath(file_name)

    # Obtaining the absolute path of the directory that will contain
    # the file. Absolute paths are used because relative paths may not
    # work with `os.path.dirname`, especially if `file_name` contains
    # no path separator (e.g. `\` in Windows and `/` in MacOS and
    # Unix).
    abs_dir = os.path.dirname(abs_path_name)

    # Creating the directory that will contain the file, if that
    # directory does not yet exist. The `exist_ok = True` argument
    # allows `os.makedirs` to work even if some subdirectory
    # components of the path in `abs_dir` already exist.
    os.makedirs(abs_dir, exist_ok = True)

def simulate_data(sigma_model_func,
                  epsilon_p_max,
                  epsilon_p_dot,
                  T_init,
                  theta_model,
                  beta_TQ,
                  rho,
                  specific_heat_func,
                  curve_size):
    """Create simulated data to test a flow stress model

    This function creates data points for a stress-strain curve while
    accounting for the temperature rise as the strain increases.

    Parameters
    ----------

    sigma_model_func : callable
        Function representing a strength model that returns the flow
        stress and takes four arguments: plastic strain, plastic
        strain rate, temperature, and some data structure containing
        the model parameters (such as an Python dictionary)

    epsilon_p_max : float
        Largest plastic strain for which stresses will be calculated

    epsilon_p_dot : float
        Plastic strain rate

    T_init : float
        Initial temperature of the sample being deformed

    theta_model
        Model parameters of the strength model

    beta_TQ : float
```

```
    Taylor-Quinney coefficient

rho : float
    Density of sample being deformed

specific_heat_func
    Function that returns the specific heat for a given
    temperature

curve_size : int
    Number of data points in the stress-strain curve

Returns
-------

dict
    A dictionary with the following keys and their associated
    values:

    - "T", a vector of the temperatures for the data points in the
      stress-strain curve
    - "epsilon_p", the plastic strains for the data points in the
      stress-strain curve
    - "sigma", the stresses for the data points in the
      stress-strain curve

"""

#!{sndepstart}
epsilon_p = np.linspace(0.0, epsilon_p_max, curve_size)
#!{sndepend}

#!{sndinitemptystart}
T = np.empty(curve_size)
sigma = np.empty(curve_size)
#!{sndinitemptyend}

#!{sndsetfirstelemstart}
T[0] = T_init

sigma[0] = sigma_model_func(
    epsilon_p[0],
    epsilon_p_dot,
    T[0],
    theta_model
)
#!{sndsetfirstelemend}

#!{sndsetotherelemsstart}
for i in range(1, curve_size):
    # Estimate of area under stress-strain curve from
    # epsilon_p[i-1] to epsilon_p[i-1].
    area_under_curve = sigma[i-1]*(epsilon_p[i] - epsilon_p[i-1])
```

```
            T_rise = beta_TQ*area_under_curve/(
                rho*specific_heat_func(T[i-1]))

            T[i] = T[i-1] + T_rise

            sigma[i] = sigma_model_func(
                epsilon_p[i],
                epsilon_p_dot,
                T[i],
                theta_model
            )
        #!{sndsetotherelemsend}

        #!{sndreturnstart}
        return {"T" : T,
                "epsilon_p": epsilon_p,
                "sigma": sigma
        }
        #!{sndreturnend}

    def plot_stress_strain_curves(output_file,
                                  epsilon_p_dot,
                                  T_init,
                                  epsilon_p, sigma,
                                  space_for_legend = 0.3,
                                  marker_period = 1,
                                  sigma_unit = "MPa",
                                  epsilon_p_dot_time_unit = "s",
                                  T_init_unit = "K",
                                  epsilon_p_label = "$\epsilon_p$",
                                  sigma_label = "$\sigma$",
                                  epsilon_p_dot_label = "$\dot{\epsilon}_p$",
                                  T_init_label = "$T_{init}$"):
        """Plots stress-strain curves

        Parameters
        ----------

        output_file : str
            Name of the file containing the plots, which may have a file
            extension `.pdf`, `.png`, `.eps`, `.ps`, or `.svg`.

        epsilon_p_dot : array_like, shape(M,)
            List or array of length M, where element `i` contains the
            strain rate for curve `i`

        T_init : array_like, shape(M,)
            List or array of length M, where element `i` contains the
            initial sample temperature for curve `i`

        epsilon_p : list of 1-d array_like
            Strain values for all curves, where `epsilon_p[0]` contains
            strain values for the first curve, `epsilon_p[1]` contains
            strain values for the second curve, etc.
```

```
sigma :  list of 1-d array_like
    Stress values for all curves, where `sigma[0]` contains stress
    values for the first curve, `sigma[1]` contains stress values
    for the second curve, etc.

space_for_legend : float, optional
    Number between zero and one indicating the amount of space in
    the plot at the bottom for the legend, such that, for example,
    space_for_legend = 0.25 increases the vertical size of the
    plot by 25%

marker_period : int, optional
    Indicates what data points may be plotted. If the data points
    are so densely spaced as to overlap with each other, then one
    may wish to set `marker_period` to a value larger than 1 in
    order to reduce the number of points shown.

sigma_unit : str
    Units of stress, e.g., "MPa". Contents of string are anything
    acceptable by matplotlib in a label, including LaTeX-like math
    notation delimited by '$'.

epsilon_dot_time_unit : str
    Units of time used in the strain rate, e.g., "s" or "sec"
    for a strain per second. Contents of string are
    anything acceptable by matplotlib in a label, including
    LaTeX-like math notation delimited by '$'.

temperature_unit : str
    Units of temperature, e.g., "K" for Kelvin.  Contents of
    string are anything acceptable by matplotlib in a label,
    including LaTeX-like math notation delimited by '$'.

epsilon_label : str, optional
    Label used for strain.  Contents of string are anything
    acceptable by matplotlib in a label, including LaTeX-like math
    notation delimited by '$'.

sigma_label : str, optional
    Label used for stress.  Contents of string are anything
    acceptable by matplotlib in a label, including LaTeX-like math
    notation delimited by '$'.

epsilon_dot_label : str, optional
    Label used for strain rate.  Contents of string are anything
    acceptable by matplotlib in a label, including LaTeX-like math
    notation delimited by '$'.

T_init_label : str, optional
    Label used for initial sample temperature.  Contents of string
    are anything acceptable by matplotlib in a label, including
    LaTeX-like math notation delimited by '$'.
```

```
"""

# The number of stress-strain curves, num_curves, should be the
# same as the number of elements in epsilon_p_dot.
num_curves = len(epsilon_p_dot)

# The markers that will be used to decorate points on the
# stress-strain curve.
markers = ["None", "o", "v", "^", "<", ">",
           "1", "2", "3", "4", "8", "s", "p", "P",
           "*", "h", "H", "+", "x", "X", "D", "d"]

# The styles of the lines that may be used in the plot.
linestyles = ["solid", "dashed", "dashdot", "dotted"]

# This determines the combinations of markers and line styles that
# will appear in the plots of successive curves. Here, the first
# curve will have marker style "None" (i.e., no marker) and a
# solid line, the next curve will also have marker style "None"
# but a dashed line, the *fifth* curve will have marker style "o"
# and a solid line, the *sixth* curve will also marker style "o"
# and a dashed line, and so on.
marker_line_combo = itertools.product(markers, linestyles)

# This creates a new empty plot figure that is 7 inches by 7
# inches.
plt.figure(figsize=(7.0, 7.0))

# This plots each of the stress-strain curves in turn
for curve_ind in range(num_curves):

    # This retrieves the next available marker and linestyle
    marker, linestyle = next(marker_line_combo)

    # This plots an individual stress-strain curve. Note that
    # epsilon_p and sigma are lists of arrays, so that
    # epsilon_p[curve_ind] and sigma[curve_ind] are the strains
    # and stresses for the current stress-strain curve.
    plt.plot(
        epsilon_p[curve_ind],
        sigma[curve_ind],
        linestyle = linestyle,
        marker = marker,

        markersize = 3, # This reduces the size of the marker in
                        # order to make the plot look nicer.

        markevery = marker_period, # This causes a marker to be
                                   # printed every "marker_period"
                                   # along the stress-strain
                                   # curve. If there are so many
                                   # data points that the markers
                                   # would overlap if
                                   # marker_period = 1, then
```

```
                                        # marker_period can be set to a
                                        # higher value to reduce the
                                        # number of markers printed.

                # This is the label to be used in the legend for this
                # curve.
                label="{} = {} {}, {} = {}/{}".format(
                    T_init_label,
                    T_init[curve_ind],
                    T_init_unit,
                    epsilon_p_dot_label,
                    epsilon_p_dot[curve_ind],
                    epsilon_p_dot_time_unit
                )
            )

        # Making room for the legend at the bottom of the plot
        ymin, ymax = plt.ylim()
        ymin -= space_for_legend*(ymax - ymin)
        plt.ylim(ymin = ymin)

        # This adds labels to the x- and y- axes.
        plt.xlabel(epsilon_p_label)
        plt.ylabel("{} ({})".format(sigma_label, sigma_unit))

        # This adds a legend to the plot
        plt.legend(
            loc="lower right", # This is the position of the legend in the
                               # plot.
            ncol = (2 if num_curves > 4 else 1) # This says that if the
                                                # number of stress-strain
                                                # curves is greater than
                                                # 4, two columns will be
                                                # used for the legend.
        )

        _ensure_path_to_file_exists(output_file)

        # This saves the plot to a file named "output_file".
        plt.savefig(output_file)

    def gen_lin_interp_func(tab_data_file,
                            x_col = 0,
                            y_col = 1,
                            conv_func_x = lambda x: x,
                            conv_func_y = lambda y: y,
                            **kwargs):
        """Generate a function that linearly interpolates tabular data

        Parameters
        ----------

        tab_data_file
            File with tabular data
```

```
    x_col
        Column of the tabular data that contains the x-values,
        defaults to 0

    y_col
        Column of the tabular data that contains the y-values,
        defaults to 1
    conv_func_x
        Function applied to all x-values in the tabular data, defaults
        to the identity function

    conv_func_y
        Function applied to all y-values in the tabular data, defaults
        to the identity function

    **kwargs

         Keyword arguments for the `pandas.read_table` function, which
         is used to read the tabular data

    Returns
    -------

    A function that returns a linear interpolation of y-values given
    an x-value
    """

    xydata = pandas.read_table(tab_data_file, **kwargs)

    return scipy.interpolate.interp1d(
        conv_func_x(xydata.iloc[:, x_col]),
        conv_func_y(xydata.iloc[:, y_col]),
        fill_value = "extrapolate"
    )

def read_from_json_file(json_file_name):
    """Reads an object from a possibly Gzip-compressed JSON file

    Parameters
    ----------
    json_file_name : str
        Name of JSON file. If the file ends in ".gz", it is assumed
        to be Gzip-compressed.

    Returns
    -------
    The object stored in the JSON file
    """

    # Again, if the file ends in ".gz", it is expected to be
    # Gzip-compressed.
    if json_file_name.endswith(".gz"):
        my_open = gzip.open
    else:
```

```
        my_open = open

    # Reading from the JSON file
    with my_open(json_file_name, "rt") as f:
        obj = json.load(f)

    return obj

def read_from_pickle_file(pkl_file_name):
    """Reads an object from a possibly Gzip-compressed pickle file

    Parameters
    ----------
    pkl_file_name : str
        Name of pickle file. If the file ends in ".gz", it is assumed
        to be Gzip-compressed.

    Returns
    -------
    The object stored in the pickle file
    """

    # Again, if the file ends in ".gz", it is expected to be
    # Gzip-compressed.
    if pkl_file_name.endswith(".gz"):
        my_open = gzip.open
    else:
        my_open = open

    # Reading from the pickle file
    with my_open(pkl_file_name, "rb") as f:
        obj = pickle.load(f)

    return obj

def save_to_pickle_file(obj, pkl_file_name):
    """Saves an object to a possibly Gzip-compressed pickle file

    Parameters
    ----------

    obj
        Any object that can be pickled
    pkl_file_name : str
        Name of pickle file. If the file ends in ".gz", it will be
        Gzip-compressed.
    """

    _ensure_path_to_file_exists(pkl_file_name)

    # Again, if the file ends in ".gz", it will be Gzip-compressed.
    if pkl_file_name.endswith(".gz"):
        my_open = gzip.open
    else:
```

```
        my_open = open

    # Writing the pickle file
    with my_open(pkl_file_name, "wb") as f:
        pickle.dump(obj, f)

if _pystan_available:
    def print_stan_summary(fit):
        """Prints summary statistics along with diagnostics

        This function mainly exists to account for two issues present
        in the release versions of PyStan available at the time of
        writing: a bug that may cause PyStan to print spurious NaN
        values in one of its diagnostics, and certain diagnostic
        checks being missing in PyStan but present in RStan. Both of
        these issues have been fixed in what is, at the time of
        writing, the current development version of PyStan.

        Parameters
        ----------

        fit : StanFit4Model
            Object containing results from a PyStan MCMC run

        """

        # Workaround for a PyStan bug
        #!{psswkaroundstart}
        try:
            pystan.constants.EPSILON = float("-inf")
        except:
            # If there is an exception, the constant is longer in
            # PyStan, so the workaround is not needed.
            pass
        #!{psswkaroundend}

        # The above constant, `pystan.constants.EPSILON`, is used in
        # PyStan as part of an attempt to avoid a divide-by-zero
        # error. When a denominator used to calculate a diagnostic called
        # `Rhat` is less than this constant, it is presumed to be
        # effectively zero, and the diagnostic value is set to
        # NaN. Unfortunately, this has led to small but still valid
        # denominator values to be spuriously treated as zero. To fix this
        # problem, `pystan.constants.EPSILON` is changed from its original
        # value of 0.000001 to negative infinity.

        #!{psssumstart}
        # Check for divergent transitions
        stan_utility.check_div(fit)

        # Check if transitions hit maximum treedepth
        stan_utility.check_treedepth(
            fit,
            max_depth = int(fit.stan_args[0]["ctrl"]["sampling"]["max_treedepth"])
```

```
    )

    # Check if BFMI is low
    stan_utility.check_energy(fit)

    # Print summary statistics
    print(fit)
    #!{psssumend}

def save_stan_fit_to_csv(fit,
                         summary_csv_filename,
                         samples_csv_filename):
    """Save summary statistics and MCMC samples to CSV files

    Parameters
    ----------

    fit : StanFit4Model object
        Object containing results from a PyStan MCMC run

    summary_csv_filename : str
        Name of CSV file to which summary statistics will be written

    samples_csv_filename : str
        Name of CSV file to which MCMC samples will be written. If
        the file ends in ".gz", it will be Gzip-compressed.
    """

    #!{ssftcpathstart}
    _ensure_path_to_file_exists(summary_csv_filename)
    _ensure_path_to_file_exists(samples_csv_filename)
    #!{ssftcpathend}

    # Writing the summary to a CSV file
    #!{ssftcpathsumwritestart}
    summary = fit.summary()

    summary_df = pandas.DataFrame(summary["summary"],
                                  index = summary["summary_rownames"],
                                  columns = summary["summary_colnames"])

    summary_df.to_csv(summary_csv_filename)
    #!{ssftcpathsumwriteend}

    # Again, if the file for the MCMC samples ends in ".gz", it
    # will be Gzip-compressed.
    #
    #!{ssftcpathgzipstart}
    if samples_csv_filename.endswith(".gz"):
        my_open = gzip.open
    else:
        my_open = open
    #!{ssftcpathgzipend}
```

```
# When permuted = False, fit.extract() returns a
# three-dimensional array. The size of the first dimension is
# the number of iterations; the size of the second dimension
# is the number of chains; the size of the third dimension is
# the number of parameters, including the pseudoparameter
# `lp__`. The samples from chain `chain_id` associated with
# the parameter named `summary["summary_rownames"][i]` are in
# `samples[:, chain_id, i]`.
#
#!{ssftcpathsampwritestart}
samples = fit.extract(permuted = False)
#!{ssftcpathsampwritemid1}

# Writing the samples to a CSV file

# Due to a quirk of NumPy's savetxt function, the CSV file has to be
# opened in binary mode (the "b" in "wb"), not text mode, even though
# the file is text. (See https://github.com/numpy/numpy/issues/6356.)
#
#!{ssftcpathsampwritemid2}

with my_open(samples_csv_filename, "wb") as f:
    #!{ssftcpathsampwritemid3}

    # Since `summary["summary_rownames"]` contains the names
    # of the parameters in their proper order, it's used for
    # the column headers of the CSV file.
    #
    #!{ssftcpathsampwritemid4}

    csv_header = ",".join(summary["summary_rownames"])
    #!{ssftcpathsampwritemid5}

    # The "encode("utf-8")" bit is needed because the file
    # is opened in binary mode.
    #
    #!{ssftcpathsampwritemid6}
    f.write("{}\n".format(csv_header).encode("utf-8"))
    #!{ssftcpathsampwritemid7}

    # Since Python uses 0-based indexing, samples.shape[1] is the size
    # of the *second* dimension of "samples".
    #
    #!{ssftcpathsampwritemid8}

    for chain_id in range(samples.shape[1]):
        #!{ssftcpathsampwritemid9}

        # As mentioned above, the samples from chain
        # `chain_id` associated with the parameter named
        # `summary["summary_rownames"][i]` are in `samples[:,
        # chain_id, i]`. Accordingly, the 2-d array with the
        # samples from all the parameters (where the
        # parameters have the same order as in
```

```
                # `summary["summary_rownames"][i]`) is `samples[:,
                # chain_id, :]`. Since the first argument to
                # np.savetxt is a file object `f` instead of a string
                # representing a file *name*, np.savetxt appends to
                # any pre-existing contents of `f`.
                #
                #!{ssftcpathsampwritemid10}
                np.savetxt(f, samples[:, chain_id, :], delimiter = ",")
        #!{ssftcpathsampwriteend}

if _pymc3_available:
    def save_pymc3_trace_to_csv(trace,
                                summary_csv_filename,
                                samples_csv_filename):
        """Save summary statistics and MCMC samples to CSV files

        Parameters
        ----------

        trace : PyMC3 trace
            Object containing results from a PyMC3 MCMC run

        summary_csv_filename : str
            Name of CSV file to which summary statistics will be written

        samples_csv_filename : str
            Name of CSV file to which MCMC samples will be written. If
            the file ends in ".gz", it will be Gzip-compressed.
        """

        #!{spttcpathstart}
        _ensure_path_to_file_exists(summary_csv_filename)
        _ensure_path_to_file_exists(samples_csv_filename)
        #!{spttcpathend}

        #!{spttcpathsumwritestart}
        pm.summary(trace).to_csv(summary_csv_filename)
        #!{spttcpathsumwriteend}

        #!{spttcpathgzipstart}
        if samples_csv_filename.endswith(".gz"):
            compression = "gzip"
        else:
            compression = None
        #!{spttcpathgzipend}

        #!{spttcpathsampwritestart}
        df = pm.trace_to_dataframe(trace)

        df.to_csv(samples_csv_filename,
                  compression = compression, index = False)
        #!{spttcpathsampwriteend}

def calc_temps(T_init, epsilon_p, sigma,
```

```
          f_area, beta_TQ, rho, specific_heat_func):
"""Calculate the temperatures for the data points along a stress-strain curve

Parameters
----------

T_init : float
    Initial temperature

epsilon_p : array_like, shape(N,)
    Sequence of plastic strains

sigma : array_like, shape(N,)
    Sequence of stresses

f_area : float
    A fraction such that f_area*sigma[0]*epsilon_p[0] is a
    reasonable estimate of the area under the missing part of the
    stress-strain curve over the interval [0, epsilon_p[0]].
    Generally, f_area should be greater than 0.5, but if
    epsilon_p[0] is zero, then f_area should be set to zero.

beta_TQ : float
    Taylor-Quinney coefficient

rho : float
    Density of sample being deformed to obtain stress-strain curve

specific_heat_func : callable
    A function accepts a temperature and returns a specific heat

Returns
-------

T : array_like, shape(N,)
    Sequence of temperatures such that T[i] is the temperature for
    data point (epsilon_p[i], sigma[i])

"""

#!{ctinitstart}
curve_size = len(epsilon_p)
T = np.empty(curve_size)

T[0] = T_init + beta_TQ*f_area*sigma[0]*epsilon_p[0]/(
    rho*specific_heat_func(T_init))
#!{ctinitend}

#!{ctcalcstart}
for i in range(1, curve_size):

    # Using trapezoid rule to estimate area under stress-strain
    # curve over interval [epsilon_p[i-1], epsilon_p[i]].
    area_under_curve = 0.5*(sigma[i-1] + sigma[i])*(
```

```
            epsilon_p[i] - epsilon_p[i-1])

        T_rise = beta_TQ*area_under_curve/(
            rho*specific_heat_func(T[i-1]))

        T[i] = T[i-1] + T_rise

    return T
    #!{ctcalcend}

def hdi(samples, cred_mass=0.95):
    """Functions for computing limits of HDI of unimodal PDFs

    Adapted from the R function HDIofMCMC in the supplementary
    material of *Doing Bayesian Data Analysis* by John K. Kruschke.

    Parameters
    ----------

    samples : 1-d array_like
        List or array of samples from an MCMC run

    cred_mass : float, optional
        Credibility mass for HDI

    Returns
    -------

    tuple
        The tuple has two elements: the minimum and maximum values of
        the HDI.
    """

    sorted_samps = np.asarray(samples).flatten()
    sorted_samps.sort()

    # This indicates the number of samples in a credible interval (or
    # "ci" for short).
    num_samps_in_ci = int(np.ceil(cred_mass*sorted_samps.size))

    # This determines the number of possible credible intervals.
    num_poss_ci = sorted_samps.size - num_samps_in_ci + 1

    # Initializing ci_width_min, which will be the width of the
    # shortest possible credible interval.
    ci_width_min = float('inf')

    # Searching through all possible credible intervals to find the
    # one with the shortest width, which will be taken to be the HDI.
    for i in range(num_poss_ci):
        curr_ci_width = sorted_samps[i + num_samps_in_ci - 1] - sorted_samps[i]

        if curr_ci_width < ci_width_min:
            ci_width_min = curr_ci_width
```

```
            ci_width_min_index = i

    return (sorted_samps[ci_width_min_index],
            sorted_samps[ci_width_min_index + num_samps_in_ci - 1])
```

C.2 Module File `jc.py`

```
def jc(epsilon_p, log_epsilon_p_dot, T_star,
       A, B, n, C, m):
    """Flow stress according to the Johnson-Cook model

    Parameters
    ----------

    epsilon_p
        Strain

    log_epsilon_p_dot
        Natural logarithm of the normalized strain rate (i.e. strain
        rate divided by the reference strain rate)

    T_star
        Normalized temperature, usually (T - T_room)/(T_melt - T_room),
        where T_melt and T_room are the melting and room temperatures

    A, B, n, C, m
        The Johnson-Cook parameters
    """

    return ((A + B*(epsilon_p**n))*
            (1.0 + C*log_epsilon_p_dot)*(1 - T_star**m))
```

C.3 Module File `jc_pymc3.py`

```
from jc import jc
import numpy as np
import pymc3 as pm

def make_jc_model(epsilon_p, sigma,
                  epsilon_p_dot, T,
                  T_melt, T_room, epsilon_p_dot_0,
                  prior_params):

    """Create a PyMC3 model conforming to the Zerilli-Armstrong (BCC) model

    Parameters
    ----------

    epsilon_p : list of 1-d NumPy array
        Strain values for all curves, where `epsilon_p[0]` contains
        strain values for the first curve, `epsilon_p[1]` contains
        strain values for the second curve, etc.

    sigma : list of 1-d NumPy array
```

```
    Stress values for all curves, where `sigma[0]` contains stress
    values for the first curve, `sigma[1]` contains stress values
    for the second curve, etc.

epsilon_p_dot : 1-d array_like
    List or array where element `i` contains the strain rate for
    curve `i`

T : list of 1-d array_like
    Temperature values for all curves, where `T[0]` contains
    temperature values for the first curve, `T[1]` contains
    temperature values for the second curve, etc.

T_melt : float
    Melting temperature

T_room : float
    Room temperature

epsilon_p_dot_0 : float
    Reference strain rate, usually 1.0 per second.

prior_params : dict
    Dictionary with the following keys: "A_guess_mean",
    "B_guess_mean", "C_guess_mean", "m_guess_mean",
    "sd_sigma_guess_mean", "A_guess_sd", "B_guess_sd",
    "C_guess_sd", "m_guess_sd", "sd_sigma_guess_sd", "n_alpha",
    and "n_beta". The values corresponding to
    "sd_sigma_guess_mean" and "sd_sigma_guess_sd" are lists or 1-d
    arrays with 2 elements, where both elements are positive
    numbers. Values corresponding to other keys are positive
    scalars.

Returns
-------

A PyMC3 model
"""

PosNormal = pm.Bound(pm.Normal, lower = 0.0)

model = pm.Model()

num_curves = len(epsilon_p)
T_melt_minus_T_room = T_melt - T_room
log_epsilon_p_dot = np.log(np.asarray(epsilon_p_dot)/epsilon_p_dot_0)

with model:

    # Priors
    A = PosNormal("A",
                  mu = prior_params["A_guess_mean"],
                  sd = prior_params["A_guess_sd"])
```

```
        B = PosNormal("B",
                      mu = prior_params["B_guess_mean"],
                      sd = prior_params["B_guess_sd"])

        n = pm.Beta("n",
                    alpha = prior_params["n_alpha"],
                    beta = prior_params["n_beta"])

        C = PosNormal("C",
                      mu = prior_params["C_guess_mean"],
                      sd = prior_params["C_guess_sd"])

        m = PosNormal("m",
                      mu = prior_params["m_guess_mean"],
                      sd = prior_params["m_guess_sd"])

        sd_sigma = PosNormal("sd_sigma",
                             mu = np.asarray(prior_params["sd_sigma_guess_mean"]),
                             sd = np.asarray(prior_params["sd_sigma_guess_sd"]),
                             shape = 2)

        for i in range(num_curves):
            T_star = (T[i] - T_room)/T_melt_minus_T_room

            pm.Normal("sigma_curve{}".format(i),
                      mu = jc(epsilon_p[i],
                              log_epsilon_p_dot[i], T_star,
                              A, B, n, C, m),
                      sd = (sd_sigma[0]
                            if (epsilon_p_dot[i] <= 1.0)
                            else sd_sigma[1]),
                      observed = sigma[i])

    return model
```

C.4 Module File `stan_utility.py`

This module file has been written by Betancourt and used in his case study on robust workflows with PyStan.[2] To comply with the 3-Clause BSD License[3] under which it is distributed, the contents of the license have been added as comments at the top of the original source file, which has not been otherwise altered.

```
# Copyright 2017 Columbia University
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
```

[2] Betancourt M. Robust PyStan workflow. c2017 [accessed 2018 Mar]. `http://mc-stan.org/users/documentation/case-studies.html#robust-pystan-workflow`.

[3] The 3-Clause BSD License. c1999 [accessed 2018 Jul]. `https://opensource.org/licenses/BSD-3-Clause`.

```
# are met:
#
# 1. Redistributions of source code must retain the above copyright
# notice, this list of conditions and the following disclaimer.
#
# 2. Redistributions in binary form must reproduce the above copyright
# notice, this list of conditions and the following disclaimer in the
# documentation and/or other materials provided with the distribution.
#
# 3. Neither the name of the copyright holder nor the names of its
# contributors may be used to endorse or promote products derived from
# this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.

import pystan
import pickle
import numpy

def check_div(fit):
    """Check transitions that ended with a divergence"""
    sampler_params = fit.get_sampler_params(inc_warmup=False)
    divergent = [x for y in sampler_params for x in y['divergent__']]
    n = sum(divergent)
    N = len(divergent)
    print('{} of {} iterations ended with a divergence ({}%)'.format(n, N,
            100 * n / N))
    if n > 0:
        print('  Try running with larger adapt_delta to remove the divergences')

def check_treedepth(fit, max_depth = 10):
    """Check transitions that ended prematurely due to maximum tree depth limit"""
    sampler_params = fit.get_sampler_params(inc_warmup=False)
    depths = [x for y in sampler_params for x in y['treedepth__']]
    n = sum(1 for x in depths if x == max_depth)
    N = len(depths)
    print(('{} of {} iterations saturated the maximum tree depth of {}'
            + ' ({}%)').format(n, N, max_depth, 100 * n / N))
    if n > 0:
        print('  Run again with max_depth set to a larger value to avoid
            saturation')

def check_energy(fit):
```

```
    """Checks the energy Bayesian fraction of missing information (E-BFMI)"""
    sampler_params = fit.get_sampler_params(inc_warmup=False)
    no_warning = True
    for chain_num, s in enumerate(sampler_params):
        energies = s['energy__']
        numer = sum((energies[i] - energies[i - 1])**2 for i in range(1, len(
            energies))) / len(energies)
        denom = numpy.var(energies)
        if numer / denom < 0.2:
            print('Chain {}: E-BFMI = {}'.format(chain_num, numer / denom))
            no_warning = False
    if no_warning:
        print('E-BFMI indicated no pathological behavior')
    else:
        print('  E-BFMI below 0.2 indicates you may need to reparameterize your
            model')

def check_n_eff(fit):
    """Checks the effective sample size per iteration"""
    fit_summary = fit.summary(probs=[0.5])
    n_effs = [x[4] for x in fit_summary['summary']]
    names = fit_summary['summary_rownames']
    n_iter = len(fit.extract()['lp__'])

    no_warning = True
    for n_eff, name in zip(n_effs, names):
        ratio = n_eff / n_iter
        if (ratio < 0.001):
            print('n_eff / iter for parameter {} is {}!'.format(name, ratio))
            print('E-BFMI below 0.2 indicates you may need to reparameterize your
                model')
            no_warning = False
    if no_warning:
        print('n_eff / iter looks reasonable for all parameters')
    else:
        print('  n_eff / iter below 0.001 indicates that the effective sample size
             has likely been overestimated')

def check_rhat(fit):
    """Checks the potential scale reduction factors"""
    from math import isnan
    from math import isinf

    fit_summary = fit.summary(probs=[0.5])
    rhats = [x[5] for x in fit_summary['summary']]
    names = fit_summary['summary_rownames']

    no_warning = True
    for rhat, name in zip(rhats, names):
        if (rhat > 1.1 or isnan(rhat) or isinf(rhat)):
            print('Rhat for parameter {} is {}!'.format(name, rhat))
            no_warning = False
    if no_warning:
        print('Rhat looks reasonable for all parameters')
```

```
    else:
        print('  Rhat above 1.1 indicates that the chains very likely have not
            mixed')

def check_all_diagnostics(fit):
    """Checks all MCMC diagnostics"""
    check_n_eff(fit)
    check_rhat(fit)
    check_div(fit)
    check_treedepth(fit)
    check_energy(fit)

def _by_chain(unpermuted_extraction):
    num_chains = len(unpermuted_extraction[0])
    result = [[] for _ in range(num_chains)]
    for c in range(num_chains):
        for i in range(len(unpermuted_extraction)):
            result[c].append(unpermuted_extraction[i][c])
    return numpy.array(result)

def _shaped_ordered_params(fit):
    ef = fit.extract(permuted=False, inc_warmup=False) # flattened, unpermuted, by
         (iteration, chain)
    ef = _by_chain(ef)
    ef = ef.reshape(-1, len(ef[0][0]))
    ef = ef[:, 0:len(fit.flatnames)] # drop lp__
    shaped = {}
    idx = 0
    for dim, param_name in zip(fit.par_dims, fit.extract().keys()):
        length = int(numpy.prod(dim))
        shaped[param_name] = ef[:,idx:idx + length]
        shaped[param_name].reshape(*([-1] + dim))
        idx += length
    return shaped

def partition_div(fit):
    """ Returns parameter arrays separated into divergent and non-divergent
        transitions"""
    sampler_params = fit.get_sampler_params(inc_warmup=False)
    div = numpy.concatenate([x['divergent__'] for x in sampler_params]).astype('
        int')
    params = _shaped_ordered_params(fit)
    nondiv_params = dict((key, params[key][div == 0]) for key in params)
    div_params = dict((key, params[key][div == 1]) for key in params)
    return nondiv_params, div_params

def compile_model(filename, model_name=None, **kwargs):
    """This will automatically cache models - great if you're just running a
    script on the command line.

    See http://pystan.readthedocs.io/en/latest/avoiding_recompilation.html"""
    from hashlib import md5

    with open(filename) as f:
```

```
        model_code = f.read()
        code_hash = md5(model_code.encode('ascii')).hexdigest()
        if model_name is None:
            cache_fn = 'cached-model-{}.pkl'.format(code_hash)
        else:
            cache_fn = 'cached-{}-{}.pkl'.format(model_name, code_hash)
        try:
            sm = pickle.load(open(cache_fn, 'rb'))
        except:
            sm = pystan.StanModel(model_code=model_code)
            with open(cache_fn, 'wb') as f:
                pickle.dump(sm, f)
        else:
            print("Using cached StanModel")
        return sm
```

C.5 Module File `za_bcc.py`

```
import numpy

def za_bcc(epsilon_p, log_epsilon_p_dot, T,
           C0, C1, C3, C4, C5, n, exp_func = numpy.exp):
    """Flow stress according to the Zerilli-Armstrong (BCC) model

    Parameters
    ----------

    epsilon_p
        Strain

    log_epsilon_p_dot
        Natural logarithm of the strain rate

    T
        Temperature

    C0, C1, C3, C4, C5, n
        The Zerilli-Armstrong (BCC) parameters. (Note that there is no
        C2 parameter, since that is for the Zerilli-Armstrong FCC
        model.)

    exp_func : function, optional
        Object representing the exponential function
    """

    return (C0 + C1*exp_func((-C3 + C4*log_epsilon_p_dot)*T) +
            C5*epsilon_p**n)
```

C.6 Module File `za_bcc_pymc3.py`

```
#!{importstart}
import numpy as np
import pymc3 as pm
from za_bcc import za_bcc
#!{importend}

def make_za_bcc_model(epsilon_p, sigma,
                      epsilon_p_dot, T,
                      prior_params):
    """Create a PyMC3 model conforming to the Zerilli-Armstrong (BCC) model

    Parameters
    ----------

    epsilon_p : list of 1-d NumPy array
        Strain values for all curves, where `epsilon_p[0]` contains
        strain values for the first curve, `epsilon_p[1]` contains
        strain values for the second curve, etc.

    sigma : list of 1-d NumPy array
        Stress values for all curves, where `sigma[0]` contains stress
        values for the first curve, `sigma[1]` contains stress values
        for the second curve, etc.

    epsilon_p_dot : 1-d array_like
        List or array where element `i` contains the strain rate for
        curve `i`

    T : list of 1-d array_like
        Temperature values for all curves, where `T[0]` contains
        temperature values for the first curve, `T[1]` contains
        temperature values for the second curve, etc.

    prior_params : dict
        Dictionary with the following keys: "C0_guess_mean",
        "C1_guess_mean", "C3_guess_mean", "C4_guess_mean",
        "C5_guess_mean", "sd_sigma_guess_mean", "C0_guess_sd",
        "C1_guess_sd", "C3_guess_sd", "C4_guess_sd", "C5_guess_sd",
        "sd_sigma_guess_sd", "n_alpha", and "n_beta". The values
        corresponding to "sd_sigma_guess_mean" and "sd_sigma_guess_sd"
        are lists or 1-d arrays with 2 elements, where both elements
        are positive numbers. Values corresponding to other keys are
        positive scalars.

    Returns
    -------

    A PyMC3 model
    """

    #!{boundnormstart}
    PosNormal = pm.Bound(pm.Normal, lower = 0.0)
```

```
#!{boundnormend}

#!{miscvarsstart}
num_curves = len(epsilon_p)
log_epsilon_p_dot = np.log(epsilon_p_dot)
#!{miscvarsend}

#!{withmodelstart}
model = pm.Model()

with model:
#!{withmodelend}

    #!{priorsstart}
    C0 = PosNormal("C0",
                   mu = prior_params["C0_guess_mean"],
                   sd = prior_params["C0_guess_sd"])

    C1 = PosNormal("C1",
                   mu = prior_params["C1_guess_mean"],
                   sd = prior_params["C1_guess_sd"])

    C3 = PosNormal("C3",
                   mu = prior_params["C3_guess_mean"],
                   sd = prior_params["C3_guess_sd"])

    C4 = PosNormal("C4",
                   mu = prior_params["C4_guess_mean"],
                   sd = prior_params["C4_guess_sd"])

    C5 = PosNormal("C5",
                   mu = prior_params["C5_guess_mean"],
                   sd = prior_params["C5_guess_sd"])

    n = pm.Beta("n",
                alpha = prior_params["n_alpha"],
                beta = prior_params["n_beta"])

    sd_sigma = PosNormal("sd_sigma",
                         mu = np.asarray(prior_params["sd_sigma_guess_mean"]),
                         sd = np.asarray(prior_params["sd_sigma_guess_sd"]),
                         shape = 2)
    #!{priorsend}

    #!{likstart}
    for i in range(num_curves):
        pm.Normal("sigma_curve{}".format(i),
                  mu = za_bcc(epsilon_p[i],
                              log_epsilon_p_dot[i], T[i],
                              C0, C1, C3, C4, C5, n,
                              exp_func = pm.math.exp),
                  sd = (sd_sigma[0]
                        if (epsilon_p_dot[i] <= 1.0)
                        else sd_sigma[1]),
```

```
                observed = sigma[i])
    #!{likend}

#!{retstart}
return model
#!{retend}
```

Appendix D. Stan Specification Files

These are the Stan specification files that have been used for Bayesian analyses of the Johnson-Cook[1] and the Zerilli-Armstrong model for body-centered cubic materials.[2] Comments in these files of the form `//!{...}` can be ignored, since they are meant to be read by tools that extract source code fragments.

D.1 Specification File `jc.stan`

```
//!{funcstart}
functions {
  vector jc(vector epsilon_p, real log_epsilon_p_dot, vector T_star,
            real A, real B, real n, real C, real m) {

    int length_epsilon_p = num_elements(epsilon_p);
    vector[length_epsilon_p] sigma;

    real edot_factor = (1.0 + C*log_epsilon_p_dot);

    // The exponentiation operator "^" doesn't vectorize, so I need a
    // "for" loop here.
    for (i in 1:length_epsilon_p) {
      sigma[i] = (A + B*(epsilon_p[i])^n)*edot_factor*
        (1.0 - (T_star[i])^m);
    }

    return sigma;
  }
}
//!{funcend}

//!{datastart}
data {
  int<lower=1> num_curves;
  int<lower=0> curve_sizes[num_curves];
  vector[num_curves] epsilon_p_dot;

  vector[sum(curve_sizes)] epsilon_p;
  vector[sum(curve_sizes)] sigma;
  vector[sum(curve_sizes)] T;

  real<lower=0.0> T_melt;
  real<lower=0.0> T_room;

  real<lower=0.0> epsilon_p_dot_0;
```

[1] Johnson GR, Cook WH. A constitutive model and data for metals subjected to large strains, high strain rates and high temperatures. In: Seventh international symposium on ballistics: Proceedings; 1983 Apr; The Hague (Netherlands). American Defense Preparedness Association; 1983. p. 541–547.

[2] Zerilli FJ, Armstrong RW. Dislocation-mechanics-based constitutive relations for material dynamics calculations. Journal of Applied Physics. 1987;61(5):1816–1825.

```
  real<lower=0.0> A_guess_mean; real<lower=0.0> A_guess_sd;
  real<lower=0.0> B_guess_mean; real<lower=0.0> B_guess_sd;
  real<lower=0.0> C_guess_mean; real<lower=0.0> C_guess_sd;
  real<lower=0.0> m_guess_mean; real<lower=0.0> m_guess_sd;

  real<lower=0.0> n_alpha; real<lower=0.0> n_beta;

  vector<lower=0.0>[2] sd_sigma_guess_mean;
  vector<lower=0.0>[2] sd_sigma_guess_sd;
}
//!{dataend}

//!{transdatastart}
transformed data {
  vector[num_curves] log_epsilon_p_dot = log(epsilon_p_dot/epsilon_p_dot_0);
  vector[sum(curve_sizes)] T_star = (T - T_room)/(T_melt - T_room);
}
//!{transdataend}

//!{paramstart}
parameters {
  real<lower=0.0> A;
  real<lower=0.0> B;
  real<lower=0.0, upper=1.0> n;
  real<lower=0.0> C;
  real<lower=0.0> m;

  real<lower=0.0> sd_sigma[2];
}
//!{paramend}

//!{modelstart}
model {
  A ~ normal(A_guess_mean, A_guess_sd)T[0.0,];
  B ~ normal(B_guess_mean, B_guess_sd)T[0.0,];
  n ~ beta(n_alpha, n_beta);
  C ~ normal(C_guess_mean, C_guess_sd)T[0.0,];
  m ~ normal(m_guess_mean, m_guess_sd)T[0.0,];

  for (i in 1:2) {
    sd_sigma[i] ~
      normal(sd_sigma_guess_mean[i],
             sd_sigma_guess_sd[i])T[0.0,];
  }

  {
    int start_ind = 1;
    for (curve_ind in 1:num_curves) {
      int end_ind = start_ind + curve_sizes[curve_ind] - 1;

      real curr_sd_sigma = (epsilon_p_dot[curve_ind] <= 1.0
                            ? sd_sigma[1]
                            : sd_sigma[2]);
```

```
      sigma[start_ind:end_ind] ~ normal(jc(epsilon_p[start_ind:end_ind],
                                           log_epsilon_p_dot[curve_ind],
                                           T_star[start_ind:end_ind],
                                           A, B, n, C, m),
                                        curr_sd_sigma);

      start_ind = end_ind + 1;
    }
  }
}
//!{modelend}
```

D.2 Specification File `za_bcc.stan`

```
functions {

  vector za_bcc(vector epsilon_p, real log_epsilon_p_dot, vector T,
                real C0, real C1, real C3, real C4, real C5, real n) {

    int length_epsilon_p = num_elements(epsilon_p);
    vector[length_epsilon_p] sigma;

    real C3_C4_fac = -C3 + C4*log_epsilon_p_dot;

    // The exponentiation operator "^" doesn't vectorize, so I need a
    // "for" loop here.
    for (i in 1:length_epsilon_p) {
      sigma[i] = C0 + C1*exp(C3_C4_fac*(T[i])) + C5*(epsilon_p[i])^n;
    }

    return sigma;
  }

}

data {
  int<lower=1> num_curves;
  int<lower=0> curve_sizes[num_curves];

  vector[num_curves] epsilon_p_dot;

  vector[sum(curve_sizes)] epsilon_p;
  vector[sum(curve_sizes)] sigma;
  vector[sum(curve_sizes)] T;

  real<lower=0.0> C0_guess_mean;
  real<lower=0.0> C0_guess_sd;

  real<lower=0.0> C1_guess_mean;
  real<lower=0.0> C1_guess_sd;

  real<lower=0.0> C3_guess_mean;
  real<lower=0.0> C3_guess_sd;
```

```
  real<lower=0.0> C4_guess_mean;
  real<lower=0.0> C4_guess_sd;

  real<lower=0.0> C5_guess_mean;
  real<lower=0.0> C5_guess_sd;

  real<lower=0.0> n_alpha;
  real<lower=0.0> n_beta;

  real<lower=0.0> sd_sigma_guess_mean[2];
  real<lower=0.0> sd_sigma_guess_sd[2];
}

transformed data {
  vector[num_curves] log_epsilon_p_dot = log(epsilon_p_dot);
}

parameters {
  real<lower=0.0> C0;
  real<lower=0.0> C1;
  real<lower=0.0> C3;
  real<lower=0.0> C4;
  real<lower=0.0> C5;
  real<lower=0.0, upper=1.0> n;

  real<lower=0.0> sd_sigma[2];
}

model {
  C0 ~ normal(C0_guess_mean, C0_guess_sd)T[0.0,];
  C1 ~ normal(C1_guess_mean, C1_guess_sd)T[0.0,];
  C3 ~ normal(C3_guess_mean, C3_guess_sd)T[0.0,];
  C4 ~ normal(C4_guess_mean, C4_guess_sd)T[0.0,];
  C5 ~ normal(C5_guess_mean, C5_guess_sd)T[0.0,];

  n ~ beta(n_alpha, n_beta);

  for (i in 1:2) {
    sd_sigma[i] ~
      normal(sd_sigma_guess_mean[i],
             sd_sigma_guess_sd[i])T[0.0,];
  }

  {
    int start_ind = 1;
    for (curve_ind in 1:num_curves) {
      int end_ind = start_ind + curve_sizes[curve_ind] - 1;

      real curr_sd_sigma = (epsilon_p_dot[curve_ind] <= 1.0
                            ? sd_sigma[1]
                            : sd_sigma[2]);

      sigma[start_ind:end_ind] ~ normal(za_bcc(epsilon_p[start_ind:end_ind],
                                               log_epsilon_p_dot[curve_ind],
```

```
                                        T[start_ind:end_ind],
                                        C0, C1, C3, C4, C5, n),
                                    curr_sd_sigma);

      start_ind = end_ind + 1;
    }
  }
}
```

β_{TQ}	Taylor-Quinney coefficient
$\dot{\epsilon}_p$	plastic strain rate
$\dot{\epsilon}_{p0}$	reference plastic strain rate, $1/s$
ϵ_p	plastic strain
ρ	density
A	fitting parameter of Johnson-Cook model that represents yield strength at reference strain rate and room temperature
B	fitting parameter of Johnson-Cook model that represents strain hardening prefactor at reference strain rate and room temperature
C	fitting parameter of Johnson-Cook model that represents strain hardening effects due to strain rate
$c(T)$	specific heat as function of temperature
C_i	fitting parameter of Zerilli-Armstrong (BCC) model, where $i \in \{0, 1, 3, 4, 5\}$
m	fitting parameter of Johnson-Cook model that represents thermal softening exponent
n	fitting parameter of Johnson-Cook and Zerilli-Armstrong models that represents strain hardening exponent
T	temperature
T^*	normalized temperature in Johnson-Cook model
T_{melt}	melting temperature
T_{room}	room temperature
1-D	one-dimensional
2-D	two-dimensional

3-D	three-dimensional
ARL	CCDC Army Research Laboratory
BCC	body-centered cubic
CSV	comma-separated value
HDI	highest density interval
IPM	interval predictor model
JSON	JavaScript Object Notation
MCMC	Markov Chain Monte Carlo
MIDAS	Material Implementation, Database, and Analysis Source
NaN	not a number
NUTS	no-U-turn sampler
PFP	pushed forward posterior
PPD	posterior predictive distribution
RHA	rolled homogeneous armor

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DIR CCDC ARL FCDD RLD CL TECH LIB
1 (PDF)	GOVT PRINTG OFC A MALHOTRA