ARL-TR-8827 • SEP 2019

Quantifying Uncertainties in Parameterizations of Strength Models of Rolled Homogeneous Armor: Part 2, R-Based Workflow

by JJ Ramsey

ARL-TR-8827 ● SEP 2019

Quantifying Uncertainties in Parameterizations of Strength Models of Rolled Homogeneous Armor: Part 2, R-Based Workflow

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

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* September 2019	**2. REPORT TYPE** Technical Report	**3. DATES COVERED (From - To)** October 2017-September 2019
4. TITLE AND SUBTITLE Quantifying Uncertainties in Parameterizations of Strength Models of Rolled Homogeneous Armor: Part 2, R-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-8827
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 Bayesian software tool RStan and the R language, 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 R package lpSolve. 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, RStan, interval predictor model

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** James J Ramsey
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	151	**19b. TELEPHONE NUMBER (Include area code)** 410-278-5614

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

1. Introduction

This report describes in detail a workflow for Bayesian analysis that uses the R language[1] and a Bayesian software tool designed to interface with that language, RStan.[2] There are two intended audiences for this report. One audience is the set of readers who have read the companion report[3] 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.[4] 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.[5] Estimation of f_{area} amounts to educated

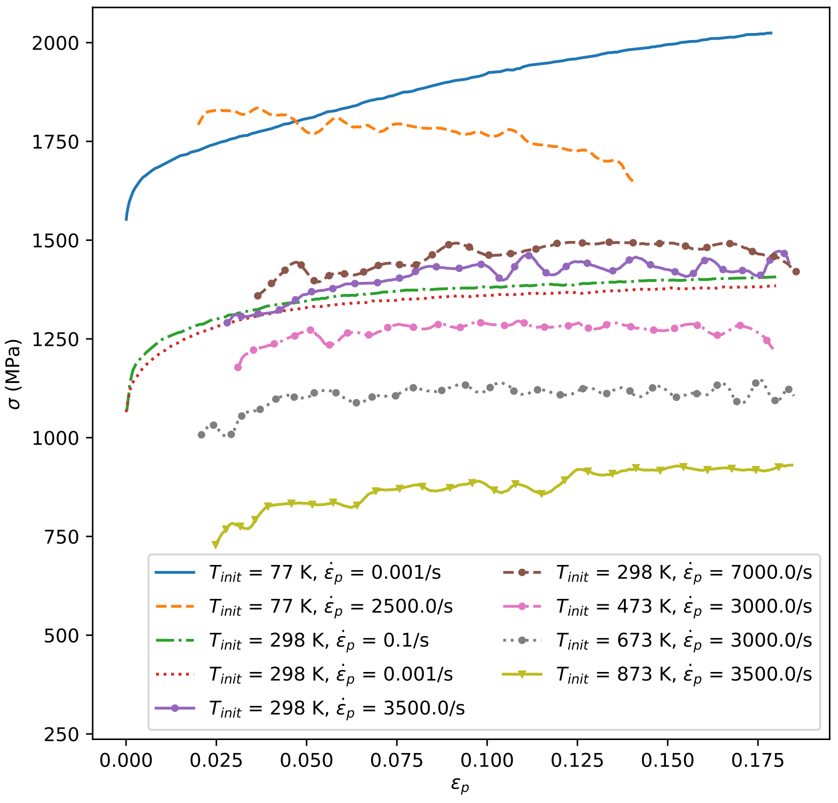


Fig. 1 Plots of flow stress σ vs. plastic strain ϵ_p for RHA from MIDAS, with the plastic strain rate denoted as $\dot{\epsilon}_p$ and the initial sample temperature T_{init}

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[6] and the Zerilli-Armstrong model for BCC materials.[7] 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.[8])

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*[9] (PPD) and the *pushed forward posterior*[10] (PFP), which can be used to check how well a model's predictions agree with the data. For the Bayesian models considered 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 tool in this report implements Markov Chain Monte Carlo (MCMC), which produces one or more chains of samples from the posterior distribution in Bayesian analysis.[9,11] The particular MCMC algorithm used is Hamiltonian Monte Carlo[12] with the no-U-turn sampler (NUTS).[13]

Finally, an alternative approach, based on an interval predictor model (IPM),[14,15] 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 $\sigma_{mdl}(\mathbf{e}, \boldsymbol{\theta}_{mdl})$ with a first-order Taylor expansion about a point estimate of $\boldsymbol{\theta}_{mdl}$, $\boldsymbol{\theta}_0$, and taking $\mathbf{\Theta}$ to be a hyperrectangle with corners $\boldsymbol{\theta}_0 - \Delta\boldsymbol{\theta}_{min}$ and $\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}_{max}$. If $\mathbf{g}_{\sigma_{mdl}}(\mathbf{e})$ is the gradient of $\sigma_{mdl}(\dots)$ with respect to $\boldsymbol{\theta}_{mdl}$ evaluated at $\mathbf{e}$ and $\boldsymbol{\theta}_0$, 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:

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

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

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

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

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

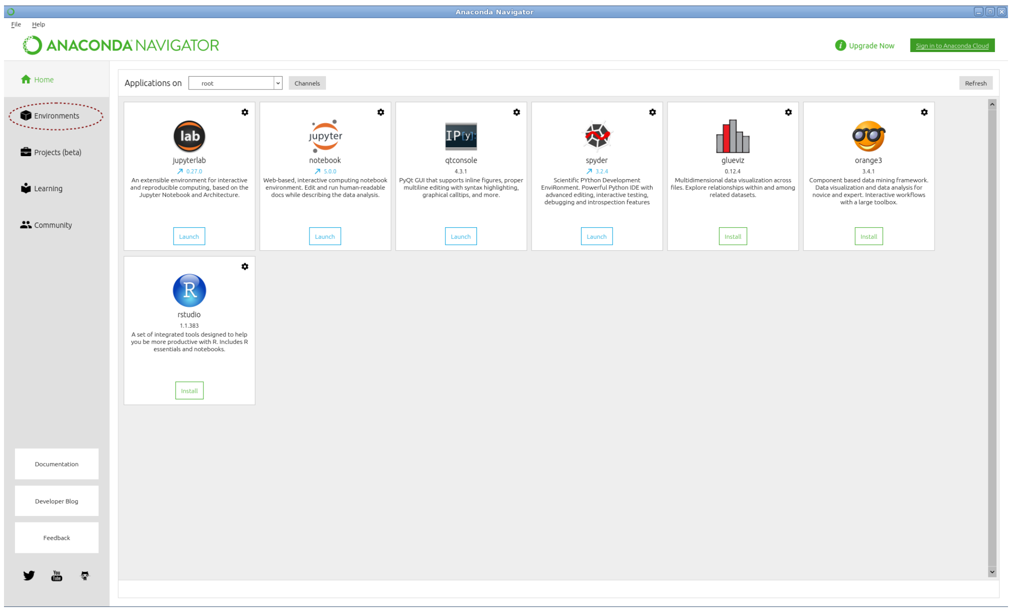
Because this report is aimed primarily at those who have had little exposure to Bayesian analysis, it is written mostly in a step-by-step tutorial style, with the R code needed for analysis shown explicitly. Readers unfamiliar with R may wish to view Appendix B. Excerpts and variables from program code, 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 RStan,[16] but here, instructions are presented for how to obtain it via the freely available Anaconda distribution.[17] (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.[18]) 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.[19] 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,[20] 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. To provide an environment for R named "Bayes", the dialog window should look like Fig. 5.

Once the environment has been created, one can then install RStan. 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. 6. 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 RStan, one can search for "rstan" and click the check box next to "r-rstan" (*not* just "rstan"!), which looks like Fig. 7.



jupyterlab
0.27.0
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.
Launch

notebook
5.0.0
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.
Launch

qtconsole
4.3.1
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.
Launch

spyder
3.2.4
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
Launch

glueviz
0.12.4
Multidimensional data visualization across files. Explore relationships within and among related datasets.
Install

orange3
3.4.1
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.
Install

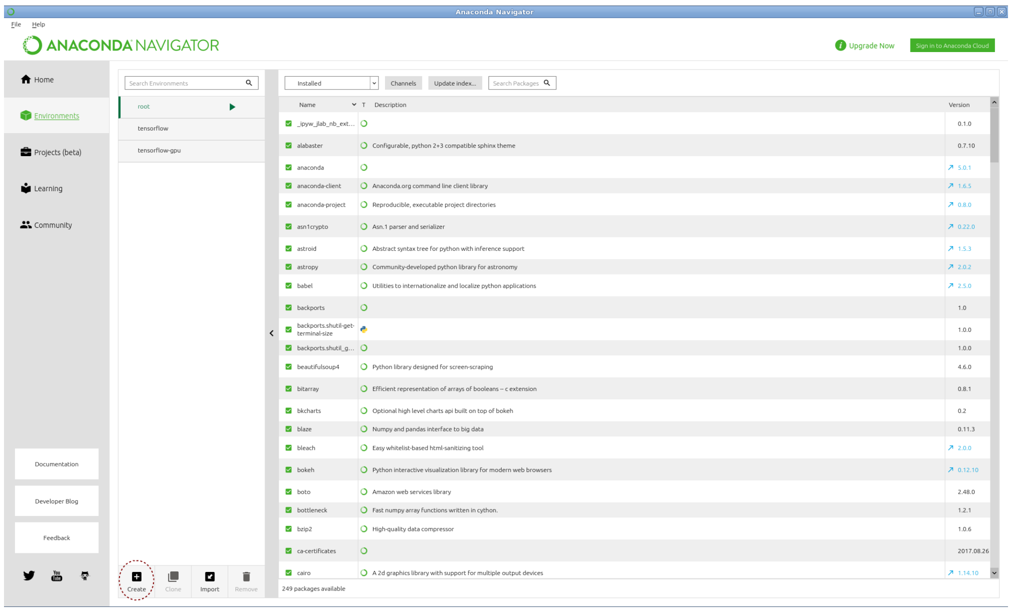
rstudio
1.1.383
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.
Install

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

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 RStan depends. To do the plotting and analysis described in later sections, one should install the packages "r-hdinterval" and "r-lpsolve" as well. This is needed for later calculations.

If one clicks on the "Home" tab in Anaconda Navigator (the tab just above the "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. 8. From this window, one can launch Jupyter Notebook,[21] where one can write and execute R code in an incremental, piecemeal fashion and intersperse the code with text explaining one's intended workflow. One can also install and then launch RStudio,[22] an integrated development environment for R.

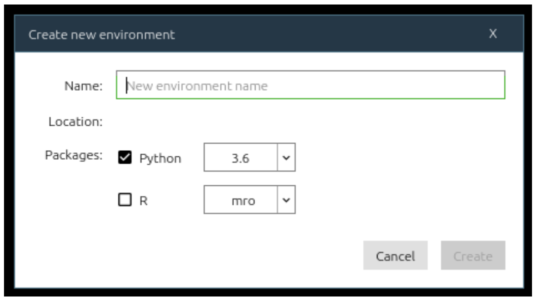
The aforementioned RStudio not only has syntax highlighting for R, but also for the Stan language.[22] For those who use the text editors Emacs or Vim, syntax highlighting for the Stan language is available as well.[23,24]



Name	T	Description	Version
✓ _ipyw_jlab_nb_ext...			0.1.0
✓ alabaster		Configurable, python 2+3 compatible sphinx theme	0.7.10
✓ anaconda			↗ 5.0.1
✓ anaconda-client		Anaconda.org command line client library	↗ 1.6.5
✓ anaconda-project		Reproducible, executable project directories	↗ 0.8.0
✓ asn1crypto		Asn.1 parser and serializer	↗ 0.22.0
✓ astroid		Abstract syntax tree for python with inference support	↗ 1.5.3
✓ astropy		Community-developed python library for astronomy	↗ 2.0.2
✓ babel		Utilities to internationalize and localize python applications	↗ 2.5.0
✓ backports			1.0
✓ backports.shutil-get-terminal-size			1.0.0
✓ backports.shutil_g...			1.0.0
✓ beautifulsoup4		Python library designed for screen-scraping	4.6.0
✓ bitarray		Efficient representation of arrays of booleans -- c extension	0.8.1
✓ bkcharts		Optional high level charts api built on top of bokeh	0.2
✓ blaze		Numpy and pandas interface to big data	0.11.3
✓ bleach		Easy whitelist-based html-sanitizing tool	↗ 2.0.0
✓ bokeh		Python interactive visualization library for modern web browsers	↗ 0.12.10
✓ boto		Amazon web services library	2.48.0
✓ bottleneck		Fast numpy array functions written in cython.	1.2.1
✓ bzip2		High-quality data compressor	1.0.6
✓ ca-certificates			2017.08.26
✓ cairo		A 2d graphics library with support for multiple output devices	↗ 1.14.10

249 packages available

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



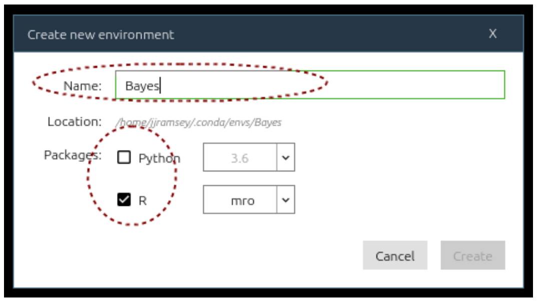
Name: New environment name

Location:

Packages: [✓] Python 3.6

[ ] R mro

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



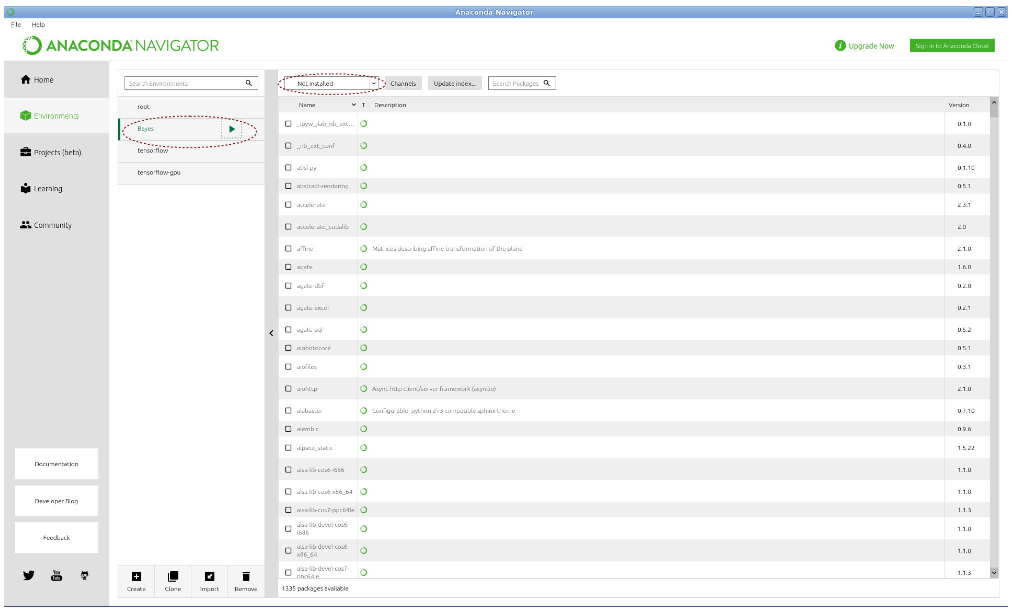
Name: Bayes

Location: /home/jiramsey/.conda/envs/Bayes

Packages: [ ] Python 3.6

[✓] R mro

Fig. 5 Dialog window for creating environments with Anaconda Navigator, with settings changed to create an environment named “Bayes” for R



Name	T	Description	Version
_ipyw_jlab_nb_ext...			0.1.0
_nb_ext_conf			0.4.0
absl-py			0.1.10
abstract-rendering			0.5.1
accelerate			2.3.1
accelerate_cudalib			2.0
affine		Matrices describing affine transformation of the plane	2.1.0
agate			1.6.0
agate-dbf			0.2.0
agate-excel			0.2.1
agate-sql			0.5.2
aiobotocore			0.5.1
aiofiles			0.3.1
aiohttp		Async http client/server framework (asyncio)	2.1.0
alabaster		Configurable, python 2+3 compatible sphinx theme	0.7.10
alembic			0.9.6
alpaca_static			1.5.22
alsa-lib-cos6-i686			1.1.0
alsa-lib-cos6-x86_64			1.1.0
alsa-lib-cos7-ppc64le			1.1.3
alsa-lib-devel-cos6-i686			1.1.0
alsa-lib-devel-cos6-x86_64			1.1.0
alsa-lib-devel-cos7-ppc64le			1.1.3

1335 packages available

Fig. 6 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.

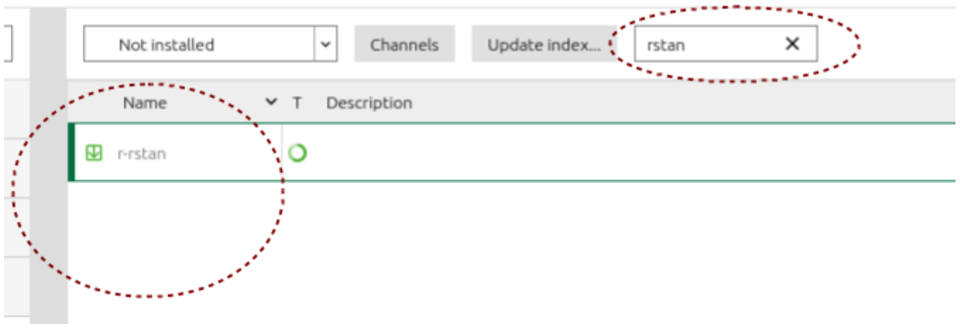
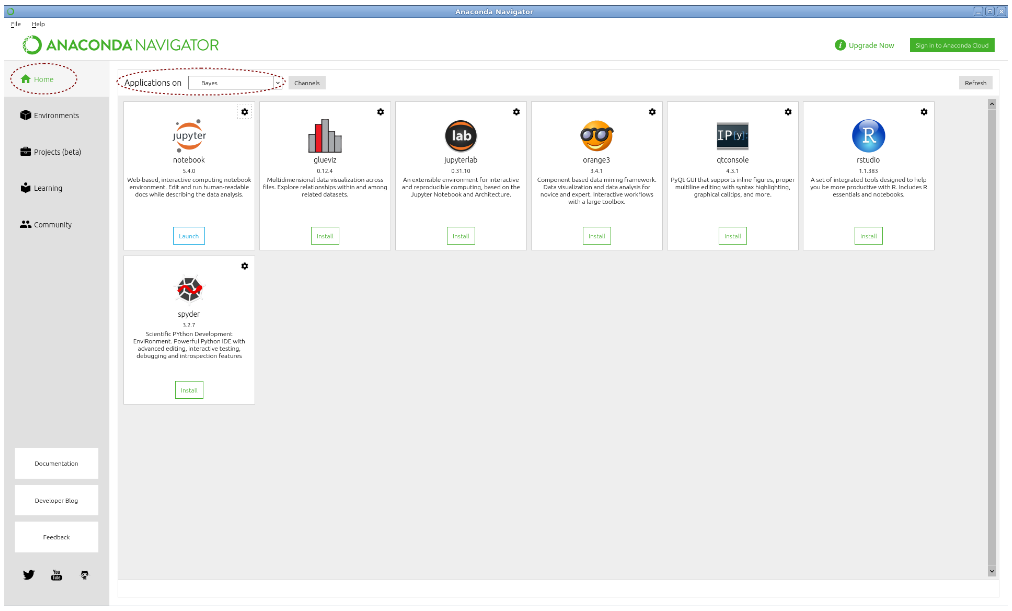


Fig. 7 Close-up of main window of Anaconda Navigator, with the "Environments" tab showing the available software package with the string "rstan" in its name. The search box and package name are circled in a dashed red line.



notebook
5.4.0
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

glueviz
0.12.4
Multidimensional data visualization across files. Explore relationships within and among related datasets.

jupyterlab
0.31.10
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

orange3
3.4.1
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

qtconsole
4.3.1
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

rstudio
1.1.383
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

spyder
3.2.7
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Fig. 8 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 `R` is the working directory from which all R code is sourced and executed. 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 $\mathrm{J/(kg \cdot K)}$. 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 Python workflow discussed in Ramsey.[25]

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.[3]

```
{
    "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,[3] 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,[4] 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 R functions, whose sources are in the file `bayes-stress-strain-utils.R` 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; and
- `save_to_rds`, a wrapper around the `saveRDS` function, which saves an R object to an R Data Serialization (RDS) file. This wrapper ensures that any directories in the file path supplied to `save_to_rds` actually exist and creates them if they do not.

The details of these functions may be mainly of interest to readers who are looking for example code to use as a reference. However, the contents of the function `simulate_data` pertain more to the physics and mathematics of the sample problem, so it is discussed in more detail. This function 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 an R list with named elements)
- `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 be and later on is generated using `gen_lin_interp_func`)
- `curve_size`, the number of data points in the stress-strain curve

The following statement creates a vector of length `curve_size`, `epsilon_p`, which contains evenly spaced strain values from 0 to `epsilon_p_max`:

```
epsilon_p <- seq(0.0, epsilon_p_max, length.out = curve_size)
```

The next two statements create the vectors `temperature` and `sigma`, which are to hold sequences of temperatures and stresses, respectively.[*] Like `epsilon_p`, these have length `curve_size`. At this point, the elements of these vectors are all zero.

```
temperature <- numeric(curve_size)
sigma <- numeric(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 `temperature` is estimated using the stress in element $i-1$ of `sigma`. To bootstrap this process, the first elements of `temperature` and `sigma` are set using the initial temperature `T_init`:

```
temperature[1] <- T_init

sigma[1] <- sigma_model_func(
  epsilon_p[1],
  epsilon_p_dot,
  temperature[1],
```

[*]In R, the variable `T` is predefined to be equivalent to the Boolean value `TRUE`, so it is not used as a variable for temperature.

```
    theta_model
  )
```

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

```
  for (i in 2:curve_size) {

    # Estimate of area under stress-strain curve from
    # epsilon_p[i-1] to epsilon_p[i].
    area_under_curve <- sigma[i-1]*(epsilon_p[i] - epsilon_p[i-1])

    temp_rise <- beta_TQ*area_under_curve/
      (rho*specific_heat_func(temperature[i-1]))

    temperature[i] <- temperature[i-1] + temp_rise

    sigma[i] <- sigma_model_func(
      epsilon_p[i],
      epsilon_p_dot,
      temperature[i],
      theta_model
    )
  }
```

As the comment in this R 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 `temperature[i]` and then `sigma[i]`.

Finally, the function returns its values as a list with named elements as follows:

```
  return (list(
      T = temperature,
      epsilon_p = epsilon_p,
      sigma = sigma
  ))
```

5.2 Testing Johnson-Cook Model with Simulated Data

First, one should load the RStan package (named `rstan`), if only to make sure it is actually there. This may be done with the following line of R code:

```
library(rstan)
```

The output of this is as follows:

```
Loading required package: ggplot2
Loading required package: StanHeaders
rstan (Version 2.17.2, GitRev: 2e1f913d3ca3)
For execution on a local, multicore CPU with excess RAM we recommend calling
options(mc.cores = parallel::detectCores()).
To avoid recompilation of unchanged Stan programs, we recommend calling
rstan_options(auto_write = TRUE)
```

One of the previous messages, the one about setting `mc.cores`, basically advises setting up RStan so that, for example, if one has at least four cores in one's CPU and one is running MCMC with, say, four chains, then the chains are generated in parallel, with one chain per core. This is done later on.

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`, shown in Appendix D. As discussed in Section 3, it is in the directory `stan_model_specs`, which is a sibling to the directory `R`, the working directory where R code is being executed. Accordingly, `stan_model_specs` is in the *parent* of the working directory, and in R, this working directory can be determined:

```
working_dir <- getwd()
```

The parent of the working directory is determined in R as follows:

```
parent_dir <- dirname(working_dir)
```

The full path to the Stan specification file `jc.stan` can then be specified as follows:

```
path_to_jc_stan_file <- file.path(parent_dir, "stan_model_specs", "jc.stan")
```

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

```
jc_model <- stan_model(path_to_jc_stan_file)
```

This compiling could have been done "behind the scenes" via the `stan` function of the RStan package. However, it is usually better to do this as an explicit step, to ensure that one has not made a syntax error in the specification. If there is such an error, then the compilation step fails, and one can then fix the specification file before proceeding to the rest of the analysis. Even if the compilation has succeeded, there may still be warnings, especially from the underlying C++ compiler that RStan 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 an RDS file. This file, named `jc.rds`, is stored in the subdirectory `compiled_stan_models`:

```
source("bayes-stress-strain-utils.R")
save_to_rds(jc_model, file.path("compiled_stan_models", "jc.rds"))
```

As mentioned before, the wrapper function `save_to_rds` ensures that the directory `compiled_stan_models` actually exists, and creates it if it does not.

One can use the `readRDS` function to bring the `stanmodel` object stored in `jc.rds` into another R session. However, an RDS file of a `stanmodel` object only works with R sessions done on the same system used to generate the RDS file, or at least a system that is nearly identical.

Simulated stress-strain curves are to be created for several combinations of plastic strain rate (`epsilon_p_dot`) and initial sample temperature (`T_init`), such as those in the following R vectors:

```
epsilon_p_dot <- c(0.001,   0.1, 3500.0, 7000.0, 3000.0, 3000.0)
T_init <-        c(298.0, 298.0,  298.0,  298.0,  473.0,  673.0)
```

These values are taken from Meyer and Kleponis.[26] They are in units of 1/s and Kelvin, respectively. To account for the temperature rise during deformation of a 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 R variable `rho`:

```
rho <- 7840.0 # kg/m^3
```

Representing the specific heat function in R is less straightforward. As mentioned in Section 4, the specific heat data in Appendix A have been collected into the CSV file `Austin_Specific_Heat_BCC_Iron.csv`. This can be read into an R session as follows:

```r
# Parent directory
parent_dir <- dirname(getwd())

c_data <- read.table(
    file.path(parent_dir, "Other_data",
              "Austin_Specific_Heat_BCC_Iron.csv"),
    sep = ","
)
```

The content of the CSV file is stored in a table-like object named `c_data`. The "`sep = ","`" argument indicates that the column entries are separated by, of course, commas. 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:

```r
# Conversion factor from MPa to Pa
MPa_to_Pa <- 1e6

c_data[,2] <- c_data[,2]/MPa_to_Pa
```

After this is done, a function that estimates the specific heat as a function of temperature can be generated as follows:

```r
c_func <- approxfun(c_data, rule = 2)
```

The previous 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 "`rule = 2`" allows for this, so that if the temperature is greater than max(`c_data[,1]`), then `c_func` returns max(`c_data[,2]`). The function `gen_lin_interp_func` from `bayes-stress-strain-utils.R` 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 an R function specifying this model. However, rather than rewrite this function from scratch, one can use the function `expose_stan_functions` to reuse the `jc` function that is already in the specification file `jc.stan`. Since `expose_stan_functions` is from RStan, the RStan package must be loaded:

```r
library(rstan)
```

The function `expose_stan_functions` needs the `stanmodel` object created from the file `jc.stan`. Since this object has already been saved to the file `jc.rds` in the directory `compiled_stan_models`, it can be brought back to the R session as follows:

```r
jc_model <- readRDS(file.path("compiled_stan_models", "jc.rds"))
```

At this point, the function `expose_stan_functions` can now be used:

```r
expose_stan_functions(jc_model)
```

Now there is a function `jc` in the R session that is essentially a copy of the one specified in `jc.stan`.

To generate the data for several simulated stress-strain curves, the function `simulate_data` (shown in Appendix C within the R source file `bayes-stress-strain-utils.R`) is used. It needs a function that not only represents the strength model but has certain arguments in a certain order. Since the `jc` function does not have these arguments in just the right form, it needs to be accessed indirectly, via a wrapper function:

```r
sigma_model_func <- function(epsilon_p,
                             epsilon_p_dot,
                             temperature,
                             theta_model) {
  log_epsilon_p_dot <-
    log(epsilon_p_dot/theta_model[["epsilon_p_dot_0"]])

  return (jc(
    epsilon_p,
    log_epsilon_p_dot,
    (temperature - 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"]]
  ))
}
```

The model parameters needed by `sigma_model_func` are shown:

```r
theta_model <- list(
  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,[26] 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 an R session. Just before generating the simulated data, the seed for the random number generator is set, so that the pseudorandom simulated data to be generated are reproducible:

```
set.seed(12345)
```

The maximum value for ϵ_p and values for $SD_{\sigma,1}$ and $SD_{\sigma,2}$ should be set at this point as well:

```
epsilon_p_max <- 0.2
sd_sigma <- c(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.

```
source("bayes-stress-strain-utils.R")

# Initializing to empty lists
sigma <- list()
epsilon_p <- list()
temperature <- list()

min_curve_size <- 40
max_curve_size <- 50

for (i in 1:length(epsilon_p_dot)) {

    if (epsilon_p_dot[i] <= 1.0) {
        beta_TQ <- 0.0
        curr_sd_sigma <- sd_sigma[1]
    } else {
        beta_TQ <- 0.9
        curr_sd_sigma <- sd_sigma[2]
    }
    # Sets curve_size to a random integer between min_curve_size
    # and max_curve_size
    curve_size <- sample(min_curve_size:max_curve_size, 1)

    curr_data <- 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[[i]] <- rnorm(n = curve_size,
                        mean = curr_data[["sigma"]],
                        sd = curr_sd_sigma)

    epsilon_p[[i]] <- curr_data[["epsilon_p"]]
    temperature[[i]] <- curr_data[["T"]]
}
```

Here, `sigma`, `epsilon_p`, and `temperature` are lists of vectors, and `sigma[[i]]` and `epsilon_p[[i]]` are the stresses and strains for stress-strain curve `i`. Furthermore, `temperature[[i]]` is a vector of temperatures, such that `temperature[[i]][j]` is the temperature for data point `j` of stress-strain curve `i`. The function `rnorm` is used to set `sigma[[i]]` to a vector of normally distributed random values, such that element `j` of the resulting vector has mean `curr_data[["sigma"]][j]` and standard deviation `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 `bayes-stress-strain-utils.R` in Appendix C). An example usage of this function is shown:

```
plot_stress_strain_curves(file.path("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. 9.

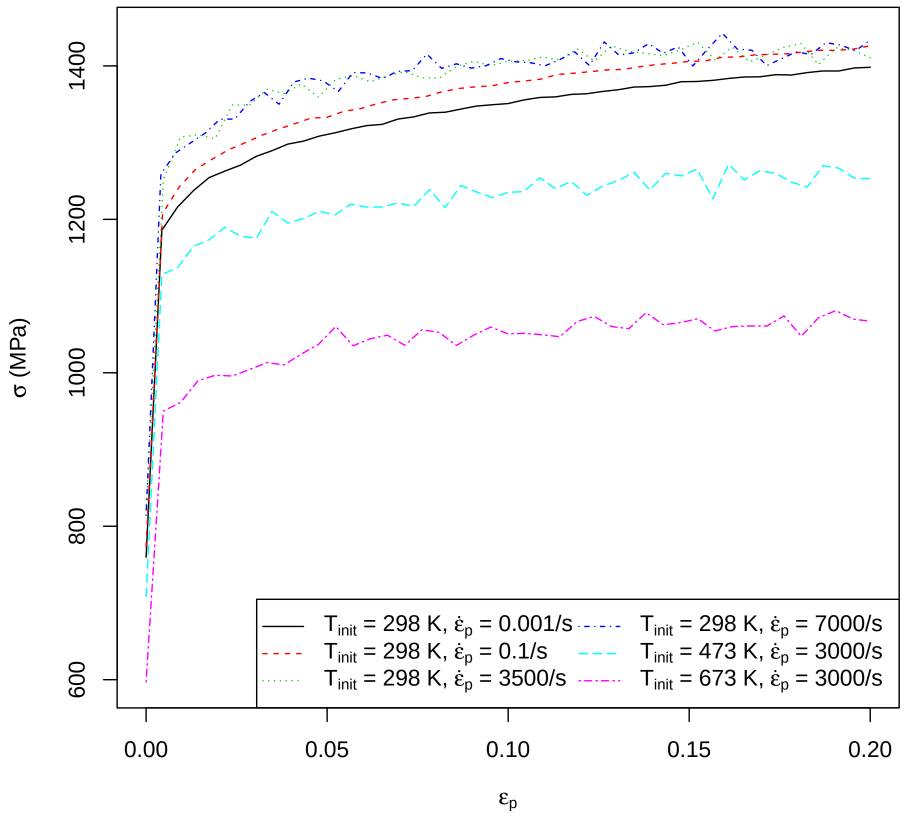


Fig. 9 Plot of simulated data used to test the Johnson-Cook RStan model

The simulated data can be saved to an RDS file in the directory `rds_data_files`, as shown. These data are saved as a list that can be used by the `sampling` function in RStan. Accordingly, variables pertaining to the priors, that is, `A_guess_mean`, `n_alpha`, and so on, are included in these data, using the JSON file `JC_priors.json` discussed in Section 4.

```
library(jsonlite)

save_to_rds(
    c(
        list(
            num_curves = length(epsilon_p_dot),
            curve_sizes = sapply(sigma, length),
            epsilon_p_dot = epsilon_p_dot,
            epsilon_p = unlist(epsilon_p),
            sigma = unlist(sigma),
            T = unlist(temperature),
            T_melt = theta_model[["T_melt"]],
            T_room = theta_model[["T_room"]],
            epsilon_p_dot_0 = theta_model[["epsilon_p_dot_0"]]
        ),
        read_json(file.path(parent_dir, "Other_data", "JC_priors.json"),
                  simplifyVector = TRUE)
    ),
    file.path("rds_data_files", "jc_simulated_data.rds")
)
```

The first argument to `save_to_rds` is an R list with named elements, where the name of each component corresponds to the name of a variable declared in the `data` program block of `jc.stan`. This list is a concatenation of two lists. In the second element of the first list, the function `sapply` applies the function `length` to each component of the list `sigma`. This is used to set the element named `curve_sizes` to a vector, such that `curve_sizes[1]` is the number of data points in the first stress-strain curve, `curve_sizes[2]` is the number of data points in the second stress-strain curve, and so on. The function `unlist` takes a list of vectors and returns one long vector, such that the first `length(sigma[[1]])` elements of `unlist(sigma)` are the elements of `sigma[[1]]`, the next `length(sigma[[2]])` elements of `unlist(sigma)` are the elements of `sigma[[2]]`, and so on.[1] 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. 10, to workaround Stan's lack of support for ragged arrays.[27] The second list is returned by a call to the function `read_json` from the `jsonlite` package.[*] For the JSON files discussed in Section 4, this function returns a list with named elements, where the names are the keys in the JSON files, and the values associated with those names are the values associated with the corresponding keys in the JSON files. The argument "`simplifyVector = TRUE`" ensures that the bracketed sequences of numbers in the JSON file are translated into vectors in R, rather than lists. This argument is needed because RStan requires that vector variables in a Stan specification file be specified with R vectors.

To run MCMC, one loads the `rstan` package, if one has not already done so, and then sets the `mc.cores` option so that chains can be generated in parallel as follows:

```
library(rstan)
```

[*]If R has been installed according to the advice in Section 2, then the `jsonlite` package should already be installed. Otherwise, it may be installed via R's `install.packages` function.

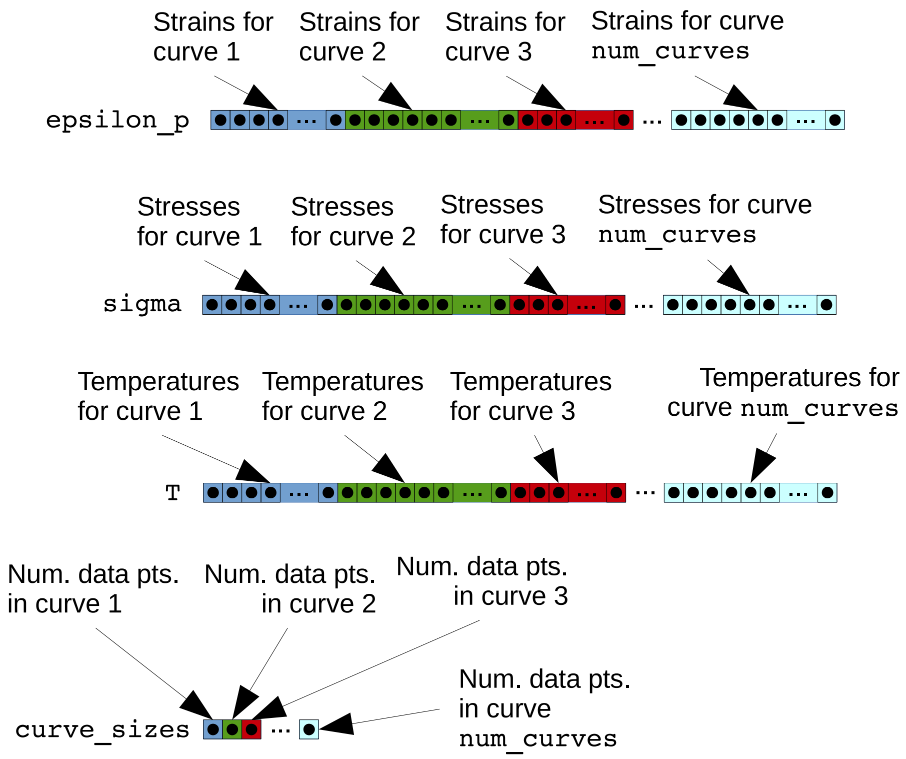


Fig. 10 Storage of data for stress-strain curves in the Stan vectors `epsilon_p`, `sigma`, `T`, and `curve_sizes`

```
options(mc.cores = parallel::detectCores())
```

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

```
jc_model <- readRDS(file.path("compiled_stan_models", "jc.rds"))

my_data <- readRDS(file.path("rds_data_files",
                             "jc_simulated_data.rds"))
```

Finally, one fits the model to the simulated data via the `sampling` function of RStan, as shown. For the sake of reproducibility, the seed for random number generation is set via the `seed` argument of the `sampling` function.

```
jc_fit <- sampling(jc_model,
                   data = my_data,
                   seed = 12345)
```

The MCMC results have been captured in a `stanfit` object named `jc_fit`. One may obtain summary statistics from this object as follows:

```
print(jc_fit, digits_summary = 6)
```

The value of `digits_summary` in the arguments of the above `print` function call indicates the number of digits shown after the decimal point in the printed statistics. The following is the output:

```
Inference for Stan model: jc.
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%
A            781.150694 0.012297 0.777720   779.696602   780.629309   781.151375
B            779.601746 0.014175 0.896483   777.921287   778.998131   779.582458
n              0.106358 0.000005 0.000297     0.105769     0.106162     0.106355
C              0.004041 0.000001 0.000034     0.003972     0.004019     0.004043
m              0.997633 0.000048 0.002614     0.992519     0.995850     0.997607
sd_sigma[1]    1.137823 0.001965 0.086759     0.982855     1.075615     1.131809
sd_sigma[2]    9.663087 0.010650 0.497499     8.748877     9.310305     9.652242
lp__        -605.183301 0.050386 1.873687  -609.733557  -606.238056  -604.852660
                    75%        97.5% n_eff     Rhat
A            781.653711   782.687873  4000 0.999645
B            780.196032   781.367302  4000 0.999574
n              0.106561     0.106942  3320 0.999728
C              0.004065     0.004106  2021 1.000398
m              0.999389     1.002844  2950 0.999645
sd_sigma[1]    1.194667     1.314741  1950 0.999858
sd_sigma[2]    9.985599    10.678908  2182 0.999745
```

```
lp__        -603.821104 -602.500861  1383 1.003835

Samples were drawn using NUTS(diag_e) at Wed Aug  1 09:55:27 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).
```

As one can see, the mean values of the parameters from the model fit are nearly the same as the parameter values in the list `theta_model` that produced the simulated data. There are also other things worth noting. First, by default, RStan 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.[27] Third, there are a couple diagnostics printed. One is the effective sample size (`n_eff`), which indicates how effectively the posterior has been sampled. The values of this 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.

RStan may or may not print the time elapsed during MCMC sampling. If it has been run from a R command prompt or RStudio, it should print it. If run from a Jupyter notebook, it probably will not. If need be, the elapsed time in seconds can be obtained as follows:

```
get_elapsed_time(jc_fit)
```

The output from the previous statement is as follows:

```
        warmup sample
chain:1   9.28   7.34
chain:2   8.92   7.83
chain:3   8.69   7.32
chain:4  10.21   7.18
```

One should note that the elapsed time may be affected by processes running in the background that do not relate to MCMC sampling.

5.3 Testing Zerilli-Armstrong (BCC) Model with Simulated Data

The process for compiling the Zerilli-Armstrong (BCC) model is nearly the same as the corresponding one for the Johnson-Cook model. One loads the RStan package, compiles the appropriate Stan specification file (i.e., `za_bcc.stan`) into a `stanmodel` object, and then saves the resulting object to an RDS file. All this is shown in the following code:

```
library(rstan)
source("bayes-stress-strain-utils.R")

parent_dir <- dirname(getwd())
za_bcc_model <- stan_model(file.path(parent_dir,
                                     "stan_model_specs",
                                     "za_bcc.stan"))

save_to_rds(za_bcc_model,
            file.path("compiled_stan_models", "za_bcc.rds"))
```

Again, the RDS file for a `stanmodel` object (here `za_bcc.rds`) only works with R sessions done on the same system used to generate the RDS file, or at least a system that is nearly identical.

Simulated stress-strain curves are to be created for several combinations of plastic strain rate (`epsilon_p_dot`) and initial sample temperature (`T_init`), such as those in the R vectors presented:

```
epsilon_p_dot <- c(2500.0, 0.001, 0.001,   0.1, 3500.0, 7000.0,
                   3000.0, 3000.0, 3500.0)
T_init <-        c(  77.0,   77.0, 298.0, 298.0,  298.0,  298.0,
                    473.0,  673.0,  873.0)
```

The values used are taken from Gray et al.[8] 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 R variable `rho` as follows. This time, the specific heat function is generated with the R function `gen_lin_interp_func` from `bayes-stress-strain-utils.R` in Appendix C.

```
source("bayes-stress-strain-utils.R")

rho <- 7840.0 # kg/m^3

# Parent directory
```

```
parent_dir <- dirname(getwd())

# Conversion factor from MPa to Pa
MPa_to_Pa <- 1e6

c_func <- gen_lin_interp_func(
    file.path(parent_dir, "Other_data",
              "Austin_Specific_Heat_BCC_Iron.csv"),
    conv_func_y = function(y) {y/MPa_to_Pa},
    sep = ","
)
```

The function `gen_lin_interp_func` encapsulates most of the previous 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 corresponds to the argument "`sep = ","`" in the call to the `read.table` function in Section 5.2.[*]

To create simulated data for the Zerilli-Armstrong (BCC) model, one needs an R function specifying this model. Rather than rewrite this function from scratch, the RStan function `expose_stan_functions` is again employed, this time to reuse the `za_bcc` function that is already in the specification file `za_bcc.stan`:

```
library(rstan)
za_bcc_model <- readRDS(file.path("compiled_stan_models", "za_bcc.rds"))
expose_stan_functions(za_bcc_model)
```

As has been done with the Johnson-Cook model, a wrapper function is used as the first argument to the function `simulate_data`:

```
sigma_model_func <- function(epsilon_p,
                             epsilon_p_dot,
                             temperature,
                             theta_model) {
  log_epsilon_p_dot <-
    log(epsilon_p_dot)

  return (za_bcc(
    epsilon_p,
    log_epsilon_p_dot,
    temperature,
    theta_model[["C0"]],
```

[*]The function `gen_lin_interp_func` actually passes "`sep = ","`" to the `read.table` function.

```
    theta_model[["C1"]],
    theta_model[["C3"]],
    theta_model[["C4"]],
    theta_model[["C5"]],
    theta_model[["n"]]
  ))
}
```

The model parameters needed by `sigma_model_func` are as follows:

```
theta_model <- list(
  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.,[8] 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 an R session. Just before generating the simulated data, the seed for the random number generator is set, so that the pseudo-random simulated data to be generated are reproducible. Also, the maximum value for ϵ_p and values for $SD_{\sigma,1}$ and $SD_{\sigma,2}$ are set as well.

```
set.seed(12345)

epsilon_p_max <- 0.2
sd_sigma <- c(1.0, 10.0)
```

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. Also, the function `rnorm` is again used to add normally distributed noise to the simulated data.

```
# Initializing to empty lists
sigma <- list()
epsilon_p <- list()
temperature <- list()

min_curve_size <- 40
max_curve_size <- 50

for (i in 1:length(epsilon_p_dot)) {
    if (epsilon_p_dot[i] <= 1.0) {
        beta_TQ <- 0.0
        curr_sd_sigma <- sd_sigma[1]
    } else {
        beta_TQ <- 0.9
        curr_sd_sigma <- sd_sigma[2]
    }

    # Sets curve_size to a random integer between min_curve_size
    # and max_curve_size
    curve_size <- sample(min_curve_size:max_curve_size, 1)

    curr_data <- 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[[i]] <- rnorm(n = curve_size,
                        mean = curr_data[["sigma"]],
                        sd = curr_sd_sigma)

    epsilon_p[[i]] <- curr_data[["epsilon_p"]]
    temperature[[i]] <- curr_data[["T"]]
}
```

Again, the simulated data is plotted to a ".pdf" file as a sanity check (via the function `plot_stress_strain_curves`), and a plot of the simulated data is shown in Fig. 11.

The simulated data are saved to an RDS file in the directory `rds_data_files`. As with the simulated data for the Johnson-Cook model, these data are saved as a list that can be used by the `sampling` function in RStan, so variables pertaining to the priors, that is, `C0_guess_mean`, `n_alpha`, and so on, are included in these data, using the JSON file `ZA_BCC_priors.json` discussed in Section 4:

```
library(jsonlite)

save_to_rds(
    c(
        list(
            num_curves = length(epsilon_p_dot),
            curve_sizes = sapply(sigma, length),
            epsilon_p_dot = epsilon_p_dot,
```

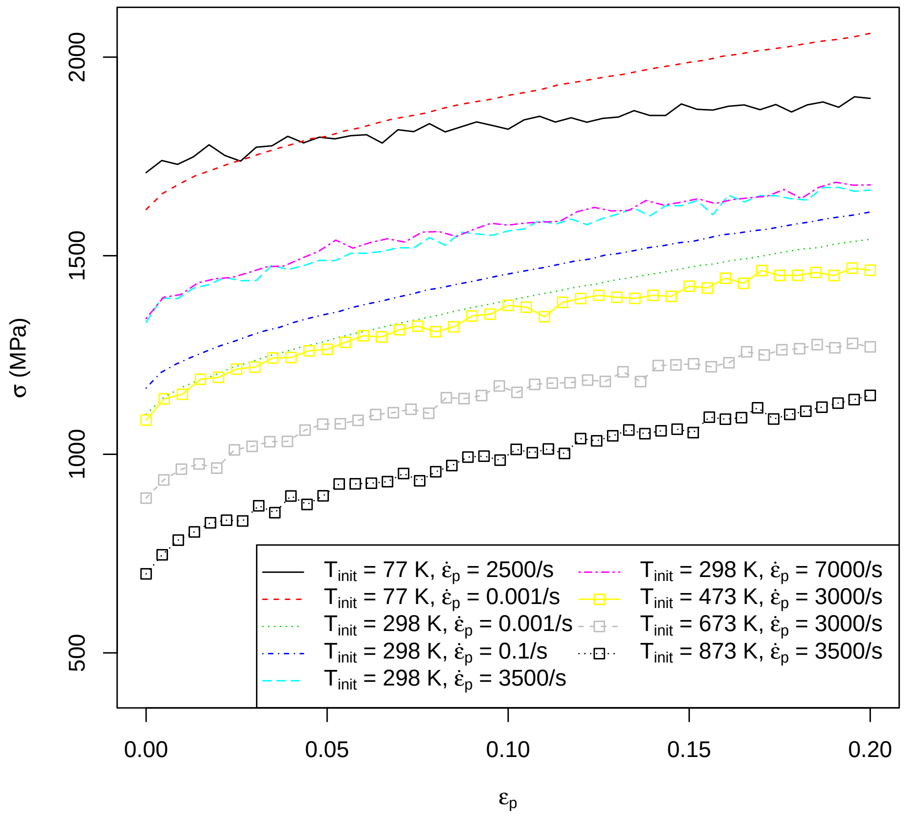


Fig. 11 Plot of simulated data used to test the Zerilli-Armstrong (BCC) RStan model

```
          epsilon_p = unlist(epsilon_p),
          sigma = unlist(sigma),
          T = unlist(temperature)
        ),
        read_json(file.path(parent_dir, "Other_data",
                            "ZA_BCC_priors.json"),
                  simplifyVector = TRUE)
    ),
    file.path("rds_data_files", "za_bcc_simulated_data.rds")
)
```

Again, to run MCMC, one loads the `rstan` package if one has not already done so, and then sets the `mc.cores` option so that chains can be generated in parallel. If it has not been loaded already, the Zerilli-Armstrong (BCC) model may be loaded into the R session, and the simulated data may be loaded as well:

```
library(rstan)
```

```r
options(mc.cores = parallel::detectCores())

za_bcc_model <- readRDS(file.path("compiled_stan_models", "za_bcc.rds"))

my_data <- readRDS(file.path("rds_data_files",
                             "za_bcc_simulated_data.rds"))
```

Much as with the Johnson-Cook model, one fits the current model to the simulated data via the `sampling` function of RStan and prints summary statistics from the fit. Again, for the sake of reproducibility, the seed for random number generation is set via the `seed` argument of the `sampling` function:

```r
za_bcc_fit <- sampling(za_bcc_model,
                       data = my_data,
                       seed = 9001)

print(za_bcc_fit, digits_summary = 6)
```

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

```
Warning message:
"There were 395 divergent transitions after warmup. Increasing adapt_delta above 0.8 may
help. See
http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup"Warning message:
"There were 33 transitions after warmup that exceeded the maximum treedepth. Increase
max_treedepth above 10. See
http://mc-stan.org/misc/warnings.html#maximum-treedepth-exceeded"Warning message:
"There were 1 chains where the estimated Bayesian Fraction of Missing Information was low.
See
http://mc-stan.org/misc/warnings.html#bfmi-low"Warning message:
"Examine the pairs() plot to diagnose sampling problems
"


Inference for Stan model: za_bcc.
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%
C0           3.583294e+01 1.479663e+01 2.198848e+01  1.36878e-01  1.487251e+01
C1           1.352065e+03 5.518025e+02 7.805874e+02  2.76663e-01  1.332188e+03
C3           2.640730e-01 3.215470e-01 4.548490e-01  1.47900e-03  1.494000e-03
C4           3.558200e-02 4.351900e-02 6.156100e-02  4.40000e-05  4.500000e-05
C5           9.031110e+02 3.685885e+02 5.213954e+02  1.45693e-01  8.973687e+02
n            5.992710e-01 2.754600e-02 3.897800e-02  5.31797e-01  5.960140e-01
sd_sigma[1]  9.087730e-01 1.198420e-01 1.761600e-01  6.18715e-01  7.688540e-01
sd_sigma[2]  7.827480e+00 2.988042e+00 4.239705e+00  5.14714e-01  6.772682e+00
lp__        -2.774392e+84 3.279273e+84 6.857107e+84 -1.31634e+85 -1.597769e+84
                    50%          75%        97.5% n_eff         Rhat
C0            43.991020    51.617534    64.112842     2     3.062880
C1          1798.972054 1806.255105 1818.237184     2   114.887451
```

```
C3             0.001503    0.264056    1.051832     2 29198.996483
C4             0.000045    0.035578    0.142215     2  7363.027577
C5          1203.235637 1204.932522 1207.826432     2   318.693626
n              0.621200    0.622333    0.624177     2    36.369721
sd_sigma[1]    0.976000    1.031423    1.130384     2     3.428795
sd_sigma[2]   10.059199   10.433392   11.109830     2    11.902843
lp__        -886.240714 -884.554105 -882.828240     4     1.643783

Samples were drawn using NUTS(diag_e) at Thu Aug  2 15:14:02 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).
```

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. While the warnings suggest increasing `adapt_delta`, the low effective sample size suggests a different approach to correct these bad results. The effective sample size indicates that the posterior is hardly even sampled, which suggests a bad starting point for the sampling. Accordingly, the fix is to start the sampling from some reasonable initial values, such as the mean values of the priors shown in the following R list:

```r
init_values <- list(
    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 RStan,[2] it says that the initial values should be either a list of lists, where each element in the outer list is a list of initial values for a chain, or a function that returns a list of initial values. Here, the first option is taken, but it is not as straightforward as it may first appear. The following is the first attempt to create the list of lists:

```r
num_chains <- 4
init_values_list <- rep(init_values, num_chains)

print(init_values_list[[1]])
```

The output from printing the first element of `init_values_list` is as follows:

```
[1] 100
```

Clearly, the first element of this new list is not a list of initial values. Instead, the first argument to `rep` has to be `list(init_values)`:

```
init_values_list <- rep(list(init_values), num_chains)

print(init_values_list[[1]])
```

The output from printing the first element of `init_values_list` is now a list of Zerilli-Armstrong coefficients, as it should be:

```
$C0
[1] 100

$C1
[1] 1000

$C3
[1] 0.001

$C4
[1] 1e-05

$C5
[1] 1000

$n
[1] 0.5
```

MCMC can now be rerun with initial values as follows. The number of chains is explicitly set to the length of `init_values_list` for the sake of consistency:

```
za_bcc_fit <- sampling(za_bcc_model,
                       data = my_data,
                       init = init_values_list,
                       chains = length(init_values_list),
                       seed = 9001)

print(za_bcc_fit, digits_summary = 6)
```

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

```
Warning message:
"There were 55 transitions after warmup that exceeded the maximum treedepth. Increase
max_treedepth above 10. See
http://mc-stan.org/misc/warnings.html#maximum-treedepth-exceeded"Warning message:
"Examine the pairs() plot to diagnose sampling problems
"
```

```
Inference for Stan model: za_bcc.
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%
C0            48.137154 0.270503 8.789566   31.206072   42.146419   48.080893
C1          1802.272924 0.258247 8.380777 1786.019051 1796.643816 1802.352187
C3             0.001499 0.000000 0.000010    0.001479    0.001492    0.001499
C4             0.000045 0.000000 0.000000    0.000044    0.000045    0.000045
C5          1204.088827 0.039429 2.027830 1200.133062 1202.705399 1204.083906
n              0.621760 0.000027 0.001336    0.619190    0.620868    0.621754
sd_sigma[1]    1.007106 0.001283 0.060465    0.897269    0.965042    1.003927
sd_sigma[2]   10.290742 0.009084 0.456301    9.469835    9.962251   10.261916
lp__        -885.726952 0.053781 2.033308 -890.579398 -886.847923 -885.372098
                    75%        97.5% n_eff     Rhat
C0            54.066546   65.211423  1056 1.000579
C1          1807.972009 1818.807290  1053 1.000578
C3             0.001506    0.001519  1056 1.000601
C4             0.000045    0.000045  1100 1.000547
C5          1205.449726 1208.058938  2645 0.999622
n              0.622680    0.624378  2418 0.999543
sd_sigma[1]    1.045130    1.134473  2222 1.000572
sd_sigma[2]   10.595305   11.225621  2523 1.001849
lp__        -884.248241 -882.764371  1429 1.004564

Samples were drawn using NUTS(diag_e) at Thu Aug  2 15:15:32 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).
```

As one can see, the mean values of the parameters from the model fit are nearly the same as the parameter values (in the list `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 by using the `control` argument of the `sampling` function:

```
za_bcc_fit <- sampling(za_bcc_model,
                       data = my_data,
                       init = init_values_list,
                       chains = length(init_values_list),
                       control = list(max_treedepth = 15),
                       seed = 9001)

print(za_bcc_fit, digits_summary = 6)
```

The following are the summary statistics from the MCMC run:

```
Inference for Stan model: za_bcc.
```

```
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%
C0           47.378450 0.263997 8.827383   29.559070   41.576275   47.698178
C1         1803.004392 0.251489 8.413823 1786.766744 1797.322923 1802.711040
C3            0.001498 0.000000 0.000010    0.001477    0.001491    0.001498
C4            0.000045 0.000000 0.000000    0.000044    0.000045    0.000045
C5         1204.103150 0.037836 2.015789 1200.211823 1202.737726 1204.067380
n             0.621778 0.000025 0.001314    0.619135    0.620932    0.621770
sd_sigma[1]    1.006778 0.001233 0.062590    0.890767    0.964071    1.003357
sd_sigma[2]   10.265720 0.008421 0.446539    9.437021    9.956084   10.243541
lp__        -885.755071 0.055410 2.104803 -890.815405 -886.870992 -885.381233
                   75%        97.5% n_eff      Rhat
C0           53.361982   64.166921  1118 1.000861
C1         1808.435729 1819.902520  1119 1.000945
C3            0.001505    0.001518  1126 1.000854
C4            0.000045    0.000045  1166 1.001249
C5         1205.435721 1208.151237  2838 1.000438
n             0.622646    0.624372  2767 1.000655
sd_sigma[1]    1.045253    1.136896  2577 1.001333
sd_sigma[2]   10.551918   11.211138  2812 1.000345
lp__        -884.248920 -882.755629  1443 1.000669

Samples were drawn using NUTS(diag_e) at Thu Aug  2 15:17:03 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).
```

RStan no longer produces the warning that it did before, and the mean values of the parameters, effective sample sizes, and potential scale reduction factors are still reasonable.

As with the Johnson-Cook model, the elapsed time in seconds can be obtained as follows:

```
get_elapsed_time(za_bcc_fit)
```

The output from the previous statement is as follows:

```
        warmup sample
chain:1 42.83  48.11
chain:2 35.15  47.57
chain:3 41.29  43.79
chain:4 36.59  45.41
```

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` and `save_to_rds`. In addition to these are `save_stan_fit_to_csv`, a function that saves the summary statistics and MCMC samples from a `stanfit` object to CSV files, and `calc_temps`, a function for estimating the temperatures at the points of a stress-strain curve. The former function involves details pertaining to the functionality of RStan, while the latter pertains to the physics of the sample problem. Accordingly, both of these functions are examined in more detail. The arguments to `save_stan_fit_to_csv` are as follows:

- `fit`, a `stanfit` object,
- `summary_csv_filename`, the name of CSV file to which summary statistics are written, and
- `samples_csv_filename`, the name of CSV file to which MCMC samples are written. If the file ends in ".gz", it is Gzip-compressed.[28]

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 creates 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 the previous statements, `ensure_path_to_file_exists`, is defined in `bayes-stress-strain-utils.R`. The details of it may be mainly of interest to readers who are looking for example code to use as a reference.

The next R statement writes the summary statistics to the file named `summary_csv_filename` as follows:

```
write.csv(summary(fit)[["summary"]], summary_csv_filename)
```

The R expression `summary(fit)` yields a list with two named elements. One of these, named `c_summary`, is a 3-D array that contains summary statistics for each of the chains from an MCMC run, while the element simply named `summary` is a matrix (i.e., a 2-D array) that has statistics for all the chains merged together. The latter is what is of interest here. This matrix object contains the row labels and column headers seen in the summary statistics that have been shown so far (e.g., "`mean`", "`sd`", parameter names, and so on) and these are in the CSV file as well.

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. The variable `out_file` is to be an object representing a file. If `gzfile(...)` is assigned to it, then it represents a Gzip-compressed[28] file. Otherwise, it represents an ordinary text file. The function `on.exit` is used to ensure that `out_file` is closed when the function `save_stan_fit_to_csv` finishes running, even if it finishes abnormally due to some problem in writing the samples to a file (such as running out of disk space).

```
if (endsWith(samples_csv_filename, ".gz")) {
  out_file <- gzfile(samples_csv_filename, "w")
} else {
  out_file <- file(samples_csv_filename, "w")
}

# Makes sure that out_file is closed, even if something goes wrong
# in write.csv.
on.exit(close(out_file))
```

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

```
write.csv(as.matrix(fit), out_file, row.names = FALSE)
```

Here, the function `as.matrix` returns a matrix where each column is a sequence of MCMC samples for a model parameter. This matrix object contains the names of the columns, 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, or `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 column names are to be the headers of the columns in the CSV file containing the MCMC samples. The argument "`row.names = FALSE`" means 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`, a vector containing the sequence of plastic strains in a given stress-strain curve
- `sigma`, a vector containing the sequence of stresses in from the same 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 the following:

```
curve_size <- length(epsilon_p)
temperature <- numeric(curve_size)

temperature[1] <- T_init + beta_TQ*f_area*sigma[1]*epsilon_p[1]/
  (rho*specific_heat_func(T_init))
```

The first statement simply sets a descriptively named variable, `curve_size`, to the length of the vector `epsilon_p`, which is the number of data points in the stress-strain curve under consideration. The next statement initializes the vector 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 2: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(temperature[i-1]))

    temperature[i] = temperature[i-1] + T_rise
  }

  return (temperature)
```

The body of the previous `for` loop 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.[29] 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 `temperature[i]`, where `i` corresponds to index j in Eq. 1) is then the sum of the temperature rise and the temperature of the previous data point $T_{j-1}^{i_c}$ (or `temperature[i-1]`).

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

6.2 Preprocessing Experimental Data

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

```
T_init_str        <- c(  "77",    "77",    "298", "298",  "298",  "298",
                        "473",  "673",  "873")
epsilon_p_dot_str <- c("0.001", "2500", "0.001", "0.1", "3500", "7000",
                       "3000", "3000", "3500")

epsilon_p <- list()
sigma <- list()

parent_dir <- dirname(getwd())

for (i in 1:length(T_init_str)) {
    csv_filename <- sprintf("T%sK_edot%s_per_s.csv",
                            T_init_str[i],
                            epsilon_p_dot_str[i])

    out_data <- read.csv(
        file.path(parent_dir, "MIDAS_data", csv_filename),
        header = FALSE
    )

    # The first column of out_data is the strain.
    epsilon_p[[i]] <- out_data[,1]
    # The second column of out_data is the stress.
    sigma[[i]] <- out_data[,2]
}
```

The previous R 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 call to the function `simulate_data` is a call to `read.csv`, which reads experimental data from a CSV file. The argument "`header = FALSE`" in the call to `read.csv` prevents R 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:

```
source("bayes-stress-strain-utils.R")

rho <- 7840.0 # kg/m^3

MPa_to_Pa <- 1e6

c_func <- gen_lin_interp_func(
    file.path(parent_dir, "Other_data",
              "Austin_Specific_Heat_BCC_Iron.csv"),
    conv_func_y = function(y) {y/MPa_to_Pa},
    sep = ","
)
```

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):

```
temperature <- list()

beta_TQ_f_area_strs <- list(
    c("0.9", "0.75"),
    c("0.9", "0.55"),
    c("0.6", "0.55"),
    c("0.9", "0.95"),
    c("0.6", "0.95")
)

epsilon_p_dot <- as.numeric(epsilon_p_dot_str)
T_init <- as.numeric(T_init_str)

for (bTQ_fA in beta_TQ_f_area_strs) {
    bTQ_fA_str <- paste(bTQ_fA[1], bTQ_fA[2], sep = ",")
    temperature[[bTQ_fA_str]] <- list()

    beta_TQ <- as.numeric(bTQ_fA[1])
    f_area  <- as.numeric(bTQ_fA[2])

    for (i in 1:length(epsilon_p_dot)) {
        if (epsilon_p_dot[i] > 1.0) {
            temperature[[bTQ_fA_str]][[i]] <-
                calc_temps(T_init[i], epsilon_p[[i]], sigma[[i]],
                           f_area, beta_TQ, rho, c_func)
        } else {
            curve_size <- length(sigma[[i]])
            temperature[[bTQ_fA_str]][[i]] <- rep(T_init[i], curve_size)
        }
    }
}
```

(For those unfamiliar with R, the function `as.numeric` converts its arguments to numeric values, so it converts a string to its corresponding number [e.g., the string `"0.9"` becomes the number 0.9] and converts a vector of strings to a vector of numbers [e.g., `c("1", "2")` becomes `c(1, 2)`]. The function `paste` concatenates its string arguments, separating each string token by the argument `sep`. For example, `paste("0.9", "0.75", sep = ",")` returns the string `"0.9,0.75"`.)

The variable `temperature` here is a list of named elements. The name of each element is a string such as `"0.9,0.75"`, where the part of the string before the comma is a value of β_{TQ} and the part after the comma is a value of f_{area}. Each element itself is a list of vectors of temperatures, with vector `i` corresponding to a strain rate `epsilon_p_dot[i]` and initial sample temperature `T_init[i]`. For high strain rates, these vectors of temperatures are calculated by the function `calc_temps` that is in `bayes-stress-strain-utils.R` 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. 12. For reference, the code for generating them is shown:

```
line_types <- rep(1:6, length.out = length(beta_TQ_f_area_strs))
color_vals <- rep(palette(), length.out = length(beta_TQ_f_area_strs))

legend_labels <- rep(NA, length(beta_TQ_f_area_strs))
```

```
for (i in 1:length(beta_TQ_f_area_strs)) {
    bTQ_fA <- beta_TQ_f_area_strs[[i]]

    legend_labels[i] <- parse(
        text = sprintf("paste(beta[TQ], ' = ', %s, ', ', f[area], ' = ', %s)",
                       bTQ_fA[1], bTQ_fA[2])
    )
}

for (i in 1:length(epsilon_p_dot)) {
    if (epsilon_p_dot[i] > 1.0) {

        out_file <- sprintf("temps_for_T_init%sK_edot%s_per_s.pdf",
                            T_init_str[i], epsilon_p_dot_str[i])

        pdf(file = file.path("plot_files", out_file),
            title = out_file,
            pointsize = 10,
            width = 3.5, height = 4)

        xlim <- range(epsilon_p[[i]])
        ylim <- NULL

        for (bTQ_fA in beta_TQ_f_area_strs) {
            bTQ_fA_str <- paste(bTQ_fA[1], bTQ_fA[2], sep = ",")

            ylim <- range(
                c(ylim, range(temperature[[bTQ_fA_str]][[i]]))
            )
        }

        ylim[2] <- ylim[2] + 0.4*(ylim[2] - ylim[1])

        # This function is from bayes-stress-strain-utils.R
        make_empty_xy_plot(xlim, ylim)

        for (j in 1:length(beta_TQ_f_area_strs)) {
            bTQ_fA <- beta_TQ_f_area_strs[[j]]
            bTQ_fA_str <- paste(bTQ_fA[1], bTQ_fA[2], sep = ",")

            lines(epsilon_p[[i]],
                  temperature[[bTQ_fA_str]][[i]],
                  lty = line_types[j],
                  col = color_vals[j])
        }

        legend("topleft",
               legend = legend_labels,
               lty = line_types,
               col = color_vals)

        title(xlab = expression(epsilon[p]), ylab = "Temperature (K)")
        dev.off()
    }
}
```

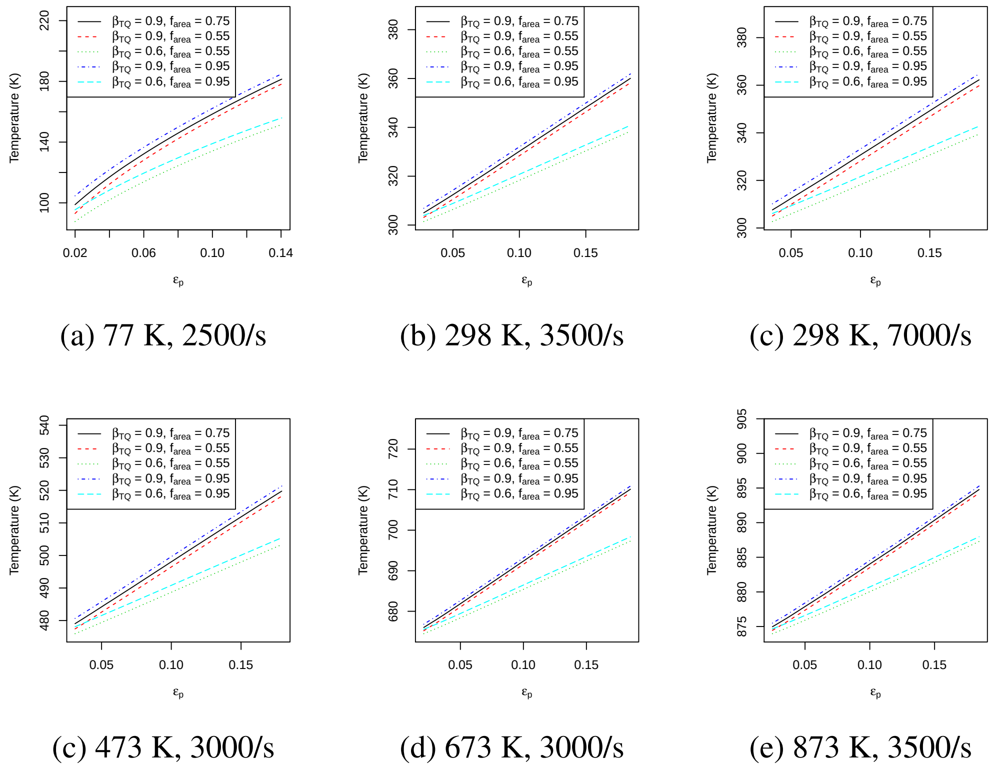


Fig. 12 Temperatures as estimated in R along stress-strain curves with the initial temperatures and strain rates shown, given the values of β_{TQ} and f_{area} in Table 1

RDS files to be used in fitting the Johnson-Cook model are saved as shown:

```
library(jsonlite)

JC_other_data <- read_json(file.path(parent_dir,
                                     "Other_data", "JC_other_data.json"))

logical_inds_JC <- (T_init >= JC_other_data[["T_room"]])

save_to_rds(
    c(
        list(
            num_curves = length(epsilon_p_dot[logical_inds_JC]),
            curve_sizes = sapply(sigma[logical_inds_JC], length),
            epsilon_p_dot = epsilon_p_dot[logical_inds_JC],
            epsilon_p = unlist(epsilon_p[logical_inds_JC]),
            sigma = unlist(sigma[logical_inds_JC])
        ),
        read_json(file.path(parent_dir, "Other_data", "JC_priors.json"),
                  simplifyVector = TRUE),
        JC_other_data
    ),
    file.path("rds_data_files", "Main_data_for_JC.rds")
)

for (bTQ_fA in beta_TQ_f_area_strs) {
    bTQ_fA_str <- paste(bTQ_fA[1], bTQ_fA[2], sep = ",")
    save_to_rds(unlist(temperature[[bTQ_fA_str]][logical_inds_JC]),
                file.path("rds_data_files",
                          sprintf("T_beta%s_farea%s_JC.rds",
                                  bTQ_fA[1], bTQ_fA[2])))
}
```

Despite appearances, this R 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 so-called *logical indices*, which are stored in the vector variable `logical_inds_JC`. These indices are used to select the components of vectors and lists that correspond to initial temperatures no less than T_{room}, since the Johnson-Cook model cannot be used with such temperatures.
- Whereas the RDS file for the simulated data includes the temperatures for points along the stress-strain curve, here the calculated temperatures are saved to separate RDS 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.rds` and the RDS file that corresponds to temperatures calculated for a particular combination of β_{TQ} and f_{area}.

Similarly, RDS 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, logical indices are not needed.

```
save_to_rds(
    c(
        list(
            num_curves = length(epsilon_p_dot),
            curve_sizes = sapply(sigma, length),
            epsilon_p_dot = epsilon_p_dot,
            epsilon_p = unlist(epsilon_p),
            sigma = unlist(sigma)
        ),
        read_json(file.path(parent_dir, "Other_data", "ZA_BCC_priors.json"),
                    simplifyVector = TRUE)
    ),
    file.path("rds_data_files", "Main_data_for_ZA_BCC.rds")
)

for (bTQ_fA in beta_TQ_f_area_strs) {
    bTQ_fA_str <- paste(bTQ_fA[1], bTQ_fA[2], sep = ",")

    save_to_rds(unlist(temperature[[bTQ_fA_str]]),
                file.path("rds_data_files",
                          sprintf("T_beta%s_farea%s_ZA_BCC.rds",
                                  bTQ_fA[1], bTQ_fA[2])))
}
```

6.3 Fitting Johnson-Cook Model to Experimental Data

After the RStan package is loaded and the `mc.cores` option is set to allow chains to be generated in parallel, the needed data for the $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ case, along with the Johnson-Cook RStan model that has been saved to an RDS file in Section 5.2, is read in as follows:

```
library(rstan)

options(mc.cores = parallel::detectCores())

my_data <- readRDS(file.path("rds_data_files",
                             "Main_data_for_JC.rds"))

my_data[["T"]] <- readRDS(file.path("rds_data_files",
                                    "T_beta0.9_farea0.75_JC.rds"))

jc_model <- readRDS(file.path("compiled_stan_models", "jc.rds"))
```

At this point, MCMC is attempted as shown in the following R code. For the sake of reproducibility, the seed for random number generation is set.

```
jc_fit <- sampling(jc_model, data = my_data, seed = 12345)
print(jc_fit, digits_summary = 6)
get_elapsed_time(jc_fit)
```

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

```
Warning message:
“There were 469 transitions after warmup that exceeded the maximum treedepth. Increase
max_treedepth above 10. See
http://mc-stan.org/misc/warnings.html#maximum-treedepth-exceeded”Warning message:
“Examine the pairs() plot to diagnose sampling problems
”
```

```
Inference for Stan model: jc.
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%
A             571.672303 1.700301 56.159711   452.989203   535.131006
B             987.192605 1.620632 53.618838   894.481225   948.319195
n               0.076875 0.000173  0.005837     0.065780     0.072850
C               0.004512 0.000002  0.000079     0.004356     0.004459
m               1.048372 0.000081  0.003614     1.041409     1.045925
sd_sigma[1]     9.367243 0.007857  0.354569     8.712622     9.121096
sd_sigma[2]    32.600706 0.014175  0.669696    31.305457    32.140418
lp__        -6807.307686 0.047835  1.919428 -6811.714616 -6808.402032
                     50%          75%        97.5% n_eff     Rhat
A             575.401206   612.454193   668.537215  1091 1.006253
B             983.440584  1021.819655  1100.312303  1095 1.006327
n               0.076763     0.080982     0.088171  1137 1.005693
C               0.004511     0.004564     0.004665  1721 1.000290
m               1.048356     1.050768     1.055583  1996 1.000536
sd_sigma[1]     9.352832     9.602424    10.090325  2037 1.000072
sd_sigma[2]    32.600814    33.058144    33.875576  2232 0.999796
lp__        -6806.967914 -6805.884931 -6804.557093  1610 1.000887

Samples were drawn using NUTS(diag_e) at Wed May 16 14:47:45 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).


        warmup sample
chain:1 330.49 379.11
chain:2 249.28 374.93
chain:3 304.57 337.48
chain:4 366.20 346.89
```

Here, MCMC has run significantly longer than in the testing run in Section 5.2, about 10 to 12 min. The potential scale reduction factors (i.e., `Rhat`) look reasonable, but there are warnings about tree depth, so MCMC is run again with `max_treedepth` set to a higher value:

```
jc_fit <- sampling(jc_model, data = my_data, seed = 12345,
                   control = list(max_treedepth = 15))
print(jc_fit, digits_summary = 6)
get_elapsed_time(jc_fit)
```

The following is the output for this new MCMC run:

```
Inference for Stan model: jc.
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%
A            576.572779 1.524483 52.744813  464.548325   541.974215
B            982.583681 1.449278 50.288671  890.056310   947.580622
n              0.077363 0.000163  0.005634    0.066654     0.073417
C              0.004519 0.000002  0.000080    0.004363     0.004464
m              1.048066 0.000076  0.003670    1.040758     1.045618
sd_sigma[1]    9.388639 0.007881  0.355841    8.716975     9.139065
sd_sigma[2]   32.563166 0.014049  0.657904   31.272480    32.127749
lp__       -6807.241852 0.046621  1.842613 -6811.520050 -6808.258301
                    50%          75%        97.5% n_eff     Rhat
A            578.755978   613.313848   674.609965  1197 1.001171
B            980.227301  1015.351992  1089.617743  1204 1.001106
n              0.077211     0.081126     0.089123  1199 1.001481
C              0.004519     0.004574     0.004669  1804 1.000601
m              1.048030     1.050534     1.055540  2338 0.999288
sd_sigma[1]    9.384069     9.618061    10.101541  2039 1.000250
sd_sigma[2]   32.558932    32.996678    33.878466  2193 1.002835
lp__       -6806.930344 -6805.880135 -6804.572277  1562 1.000945

Samples were drawn using NUTS(diag_e) at Wed May 16 15:04:50 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).


        warmup sample
chain:1 570.52 355.95
chain:2 300.54 316.61
chain:3 307.61 364.35
chain:4 530.86 295.82
```

The 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 be approximately the yield stress,[6] appears slightly low for RHA.

Nonetheless, the samples and summary from the MCMC run are saved for future examination, using the convenience function `save_stan_fit_to_csv` from the R source file `bayes-stress-strain-utils.R` in Appendix C. To save disk space, the samples are saved to a Gzip-compressed[28] CSV file:

```r
source("bayes-stress-strain-utils.R")

save_stan_fit_to_csv(
    jc_fit,
    file.path("summaries",
              "jc_MIDAS_rstan_summary_weak_prior_bTQ09_fA075.csv"),
    file.path("samples",
                 "jc_MIDAS_rstan_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, MCMC is about to be run with the strongly informative prior for A. If one starts from the same R session used for the MCMC runs with the weak prior, then only a small change to the `my_data` variable is needed:

```
library(jsonlite)

new_RHA_priors <- read_json(file.path(parent_dir,
                                      "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 both the same `seed` and `max_treedepth` values. The following is the output from this run:

```
Inference for Stan model: jc.
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%
A             699.842690 0.265203 10.100359   680.444423   692.684045
B             866.224370 0.246305  9.544459   847.425110   859.631635
n               0.092704 0.000043  0.001700     0.089416     0.091528
C               0.004542 0.000002  0.000081     0.004384     0.004486
m               1.047197 0.000073  0.003576     1.040321     1.044715
sd_sigma[1]     9.645859 0.007541  0.361472     8.958504     9.403342
sd_sigma[2]    32.346724 0.013953  0.665879    31.058152    31.886120
lp__        -6810.113182 0.045844  1.856475 -6814.504070 -6811.167520
                     50%          75%        97.5% n_eff     Rhat
A             699.903023   706.618845   719.906336  1450 1.000388
B             866.206256   872.897205   884.762912  1502 1.000436
n               0.092673     0.093788     0.096188  1589 1.000101
C               0.004543     0.004597     0.004697  2035 1.000408
m               1.047209     1.049601     1.054355  2390 1.000364
sd_sigma[1]     9.634547     9.883966    10.375682  2298 1.000577
sd_sigma[2]    32.342439    32.791396    33.659544  2278 1.000314
lp__        -6809.777220 -6808.744612 -6807.425879  1640 1.001625

Samples were drawn using NUTS(diag_e) at Wed May 30 14:37:06 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).
```

```
        warmup sample
chain:1 614.65 59.46
chain:2  93.05 62.07
chain:3 141.20 60.01
chain:4  58.93 54.01
```

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 times are mostly shorter as well, though the warmup time for the first chain is substantially longer than it is for the other chains. At this point, the results need to be saved for further analysis. The samples and summary from MCMC may be saved as follows:

```
save_stan_fit_to_csv(
    jc_fit,
    file.path("summaries",
              "jc_MIDAS_rstan_summary_strong_prior_on_A_bTQ09_fA075.csv"),
    file.path("samples",
              "jc_MIDAS_rstan_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.rds` replaced with the file for a different pair of β_{TQ} and f_{area} values, such as `T_beta0.6_farea0.95_JC.rds` for $\beta_{TQ} = 0.6$ and $f_{area} = 0.95$. The means and standard deviations of the resulting fitted parameters are in Ramsey.[3]

6.4 Fitting Zerilli-Armstrong (BCC) Model to Experimental Data

Much as with the Johnson-Cook model, after the RStan package has been loaded and the `mc.cores` option is set to allow chains to be generated in parallel, the needed data for the $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ case, along with the Zerilli-Armstrong (BCC) RStan model that has been saved to an RDS file in Section 5.3, are read in as follows:

```
library(rstan)
```

```
options(mc.cores = parallel::detectCores())

my_data <- readRDS(file.path("rds_data_files",
                             "Main_data_for_ZA_BCC.rds"))

my_data[["T"]] <- readRDS(file.path("rds_data_files",
                                    "T_beta0.9_farea0.75_ZA_BCC.rds"))

za_bcc_model <- readRDS(file.path("compiled_stan_models", "za_bcc.rds"))
```

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

```
init_vals <- list(
    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"]])
)
```

At this point, MCMC is about to be attempted. For the sake of reproducibility, the seed for random number generation is set. Also, this time, the initial values are set with a function rather than a list of lists.

```
za_bcc_fit <- sampling(za_bcc_model,
                       data = my_data,
                       init = function() {init_vals},
                       seed = 9001)
print(za_bcc_fit, digits_summary = 6)
get_elapsed_time(za_bcc_fit)
```

The output from the previous code is shown. It shows no warnings from RStan, and the effective samples sizes and potential scale reduction factors are reasonable:

```
Inference for Stan model: za_bcc.
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%
C0            107.709018 0.685993 25.574969    55.699931    90.337474
C1           1529.202699 0.195793  8.735439  1512.606233  1523.212847
C3              0.002190 0.000001  0.000032     0.002127     0.002170
C4              0.000041 0.000000  0.000001     0.000040     0.000041
C5            749.830746 0.608239 22.101167   708.437283   734.614539
n               0.158095 0.000260  0.009704     0.140086     0.151324
sd_sigma[1]    31.615321 0.018083  0.933597    29.879019    30.980362
sd_sigma[2]    47.472144 0.015622  0.854084    45.812047    46.892056
```

```
lp__        -9766.210035 0.054921  2.070180 -9771.147556 -9767.387094
                     50%          75%        97.5% n_eff     Rhat
C0             108.296876   125.361864   155.587964  1390 1.003307
C1            1528.928584  1534.829723  1546.922057  1991 1.000043
C3               0.002191     0.002211     0.002249  1715 0.999676
C4               0.000041     0.000042     0.000042  2480 1.000537
C5             749.101197   764.353075   794.479695  1320 1.002503
n                0.157975     0.164579     0.177945  1394 1.002920
sd_sigma[1]     31.593983    32.211505    33.508287  2666 1.000347
sd_sigma[2]     47.475156    48.049962    49.139706  2989 0.999948
lp__        -9765.860023 -9764.699110 -9763.200214  1421 1.000447

Samples were drawn using NUTS(diag_e) at Mon May 21 08:41:24 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).


        warmup sample
chain:1 63.76   76.19
chain:2 69.72   92.71
chain:3 68.68   75.47
chain:4 74.56  105.52
```

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

```
source("bayes-stress-strain-utils.R")

save_stan_fit_to_csv(
    za_bcc_fit,
    file.path("summaries",
              "za_bcc_MIDAS_rstan_summary_bTQ09_fA075.csv"),
    file.path("samples",
              "za_bcc_MIDAS_rstan_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 R session used for the previous fit, then an MCMC run with the new data can be done:

```
JC_main_data <- readRDS(file.path("rds_data_files",
                                  "Main_data_for_JC.rds"))
```

```
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"]] <- readRDS(file.path("rds_data_files",
                                    "T_beta0.9_farea0.75_JC.rds"))

za_bcc_fit <- sampling(za_bcc_model,
                       data = my_data,
                       init = function() {init_vals},
                       seed = 9001)
print(za_bcc_fit, digits_summary = 6)
get_elapsed_time(za_bcc_fit)
```

The following are the results from the MCMC run:

```
Warning message:
“There were 1 divergent transitions after warmup. Increasing adapt_delta above 0.8 may
help. See
http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup”Warning message:
“Examine the pairs() plot to diagnose sampling problems
”


Inference for Stan model: za_bcc.
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%
C0              1.856426 0.031029  1.827228     0.037432     0.518705
C1           1535.847623 0.343556 12.511540  1510.965623  1527.169404
C3              0.001399 0.000001  0.000022     0.001355     0.001384
C4              0.000026 0.000000  0.000001     0.000025     0.000026
C5            590.453587 0.299558 10.601781   569.280401   583.063049
n               0.178590 0.000224  0.007616     0.164242     0.173269
sd_sigma[1]    13.912729 0.013934  0.600692    12.782011    13.507633
sd_sigma[2]    43.241397 0.017583  0.905223    41.529256    42.615206
lp__        -7348.632008 0.052106  2.064795 -7353.435594 -7349.825247
                     50%          75%        97.5% n_eff     Rhat
C0              1.335665     2.592077     6.868329  3468 1.000775
C1           1535.833308  1544.301900  1560.242546  1326 1.002346
C3              0.001399     0.001414     0.001444  1178 0.999900
C4              0.000026     0.000027     0.000028  1701 0.999989
C5            590.286775   597.801277   611.037302  1253 1.001070
n               0.178315     0.183478     0.194600  1152 1.000960
sd_sigma[1]    13.883629    14.324345    15.133365  1858 1.003091
sd_sigma[2]    43.220078    43.839171    45.089457  2651 1.000596
lp__        -7348.307074 -7347.073831 -7345.609035  1570 1.000680

Samples were drawn using NUTS(diag_e) at Fri Aug  3 10:05:17 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).


        warmup sample
chain:1 44.01  46.59
chain:2 51.34  45.75
chain:3 46.18  40.33
chain:4 49.64  45.41
```

There is a warning about divergent transitions similar to the one seen in the initial test of the Zerilli-Armstrong (BCC) model in Section 5.3. However, that initial test also produced poor values for the effective sample sizes and potential scale reduction factors, something not seen in these results. Because the warning about divergences is not accompanied by indications of other problems, it is dealt with by simply following the advice shown in the warning, that is, increasing `adapt_delta`, as shown in the following R code:

```
za_bcc_fit <- sampling(za_bcc_model,
                       data = my_data,
                       init = function() {init_vals},
                       control = list(adapt_delta = 0.9),
                       seed = 9001)
print(za_bcc_fit, digits_summary = 6)
get_elapsed_time(za_bcc_fit)
```

The following are the results from the MCMC run with an increased `adapt_delta` value:

```
Inference for Stan model: za_bcc.
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%
C0              1.838311 0.032837  1.852660     0.047391     0.526852
C1           1535.879297 0.347084 12.595803  1510.485234  1527.432886
C3              0.001398 0.000001  0.000023     0.001354     0.001383
C4              0.000026 0.000000  0.000001     0.000025     0.000026
C5            590.188913 0.310604 10.787786   568.976240   582.941115
n               0.178752 0.000229  0.007809     0.163999     0.173488
sd_sigma[1]    13.927820 0.013741  0.604718    12.786078    13.516595
sd_sigma[2]    43.257861 0.018189  0.880064    41.578456    42.678689
lp__        -7348.529108 0.049524  2.010771 -7353.231987 -7349.676342
                     50%          75%        97.5% n_eff     Rhat
C0              1.267023     2.536462     6.963349  3183 1.000211
C1           1535.957295  1544.542748  1559.960558  1317 1.001718
C3              0.001397     0.001413     0.001445   972 1.000675
C4              0.000026     0.000027     0.000027  1762 1.000205
```

```
C5             589.872530   597.304725   612.356929  1206 1.001071
n                0.178528     0.183905     0.195041  1159 1.001854
sd_sigma[1]     13.909507    14.338809    15.164395  1937 1.002238
sd_sigma[2]     43.238246    43.810689    45.077101  2341 1.001283
lp__         -7348.221335 -7347.047432 -7345.618649  1649 1.002222

Samples were drawn using NUTS(diag_e) at Fri Aug  3 10:07:09 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).


        warmup sample
chain:1 51.60  47.91
chain:2 44.17  42.49
chain:3 56.44  48.90
chain:4 56.78  52.11
```

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 saved for future examination:

```r
save_stan_fit_to_csv(
    za_bcc_fit,
    file.path("summaries",
              "za_bcc_MIDAS_rstan_summary_JC_data_bTQ09_fA075.csv"),
    file.path("samples",
              "za_bcc_MIDAS_rstan_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.[3]

6.5 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`:

```r
# Loading the jc function
library(rstan)
jc_model <- readRDS(file.path("compiled_stan_models", "jc.rds"))
expose_stan_functions(jc_model)
```

```
# Defining the gradient
jc_grad <- function(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 <- ifelse(epsilon_p == 0,
                   rep(0, length(epsilon_p)),
                   B*(epsilon_p^n)*(-T_star^m + 1)*
                   (C*log_epsilon_p_dot + 1.0)*
                     log(epsilon_p))

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

    dJCdm <- ifelse(T_star == 0,
                   rep(0, length(T_star)),
                   -T_star^m*(A + B*epsilon_p^n)*(C*log_epsilon_p_dot + 1.0)*
                   log(T_star))

  return (rbind(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[30]). 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 $\epsilon_p \rightarrow 0$ and $T^* \rightarrow 0$, these factors approach zero, and the numerical calculation of the derivatives reflects that. Also, to allow the R variables `epsilon_p` and `T_star` to be arrays, `ifelse` is used rather than a raw `if` statement.

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

```
my_data <- readRDS(file.path("rds_data_files", "Main_data_for_JC.rds"))

my_data[["T"]] = readRDS(
    file.path("rds_data_files","T_beta0.9_farea0.75_JC.rds"))

epsilon_p <- my_data[["epsilon_p"]]
epsilon_p_dot <- my_data[["epsilon_p_dot"]]

log_ep_dot <- log(epsilon_p_dot/my_data[["epsilon_p_dot_0"]])

T_room <- my_data[["T_room"]]
T_star <- (my_data[["T"]] - T_room)/(my_data[["T_melt"]] - T_room)
```

```
curve_sizes <- my_data[["curve_sizes"]]
```

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

```
MPa_to_GPa <- 1e-3
sigma <- my_data[["sigma"]]*MPa_to_GPa
```

For the sake of array calculations that are to be needed, `log_ep_dot_vec` is created from `log_ep_dot` as follows:

```
log_ep_dot_vec <- NULL
for (i in 1:length(curve_sizes)) {
    log_ep_dot_vec <- c(
        log_ep_dot_vec,
        rep(log_ep_dot[i], curve_sizes[i])
    )
}
```

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:

```
summary <- read.csv(
    file.path("summaries",
              "jc_MIDAS_rstan_summary_strong_prior_on_A_bTQ09_fA075.csv"),
    row.names = 1
)

theta_0 <- c(summary["A", "mean"]*MPa_to_GPa,
             summary["B", "mean"]*MPa_to_GPa,
             summary["n", "mean"],
             summary["C", "mean"],
             summary["m", "mean"])
```

At this point, one can begin to construct the inputs that will be needed for constrained minimization with the function `lp` from the `lpSolve` R package.[31] 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) a combination of an array and 2 vectors 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(epsilon_p, log_ep_dot_vec, T_star,
                      theta_0[1], theta_0[2], theta_0[3],
                      theta_0[4], theta_0[5])
```

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 <- abs(g_sigma_mdl)
g_sigma_mdl_abs_sum <- rowSums(g_sigma_mdl_abs)
```

The vector of coefficients, then is as follows:

```
coefficients <- c(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 <- length(epsilon_p)
coefficients <- coefficients/num_data_pts
```

The inequalities in Eq. 14 need to be rearranged to fit the form needed by the function `lp`, that is, $\mathbf{Au}\ cmp\ \mathbf{b}$. 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 $\mathbf{u}$. $\mathbf{b}$ is a vector with the same number of rows as $\mathbf{A}$. The operator *cmp* is really a vector of operators, one for each row of $\mathbf{A}$ and its corresponding element in $\mathbf{b}$. Each element of *cmp* is one of $<$, $\leq$, $=$, $\geq$, or $>$. To fit this format, Eq. 14 can be combined with Eqs. 12 and 13 and rearranged to obtain

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

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

Accordingly, then, the matrix $\mathbf{A}$ and vectors $\mathbf{b}$ and *cmp* can be constructed in R as follows:

```r
num_data_pts2 <- 2*num_data_pts
num_data_pts_p1 <- num_data_pts + 1
num_coeffs = length(coefficients)

A_mat <- matrix(nrow = num_data_pts2,
                ncol = num_coeffs)

b_vec <- rep(NA, 2*num_data_pts)

g_gabs_half_sumT <- t(0.5*(g_sigma_mdl + g_sigma_mdl_abs))
g_gabs_half_diffT <- t(0.5*(g_sigma_mdl - g_sigma_mdl_abs))

half_num_coeffs <- num_coeffs/2
half_num_coeffs_p1 <- half_num_coeffs + 1

# Because jc is generated from expose_stan_functions, it doesn't
# vectorize, hence the need for mapply.
sigma_mdl <- mapply(jc, epsilon_p, log_ep_dot_vec, T_star,
                    MoreArgs = list(theta_0[1], theta_0[2], theta_0[3],
                                    theta_0[4], theta_0[5]))

sigma_minus_sigma_mdl <- sigma - sigma_mdl

A_mat[1:num_data_pts, 1:half_num_coeffs] <- -g_gabs_half_sumT
A_mat[1:num_data_pts, half_num_coeffs_p1:num_coeffs] <- g_gabs_half_diffT

cmp_vec <- rep("<=", num_data_pts)

b_vec[1:num_data_pts] <- sigma_minus_sigma_mdl

A_mat[num_data_pts_p1:num_data_pts2, 1:half_num_coeffs] <- -g_gabs_half_diffT
A_mat[num_data_pts_p1:num_data_pts2, half_num_coeffs_p1:num_coeffs] <-
    g_gabs_half_sumT

cmp_vec <- c(cmp_vec, rep(">=", num_data_pts))

b_vec[num_data_pts_p1:num_data_pts2] <- sigma_minus_sigma_mdl
```

At this point, the minimization can proceed:

```r
library(lpSolve)

result <- lp(objective.in = coefficients,
             const.mat = A_mat,
             const.dir = cmp_vec,
             const.rhs = b_vec)

cat(sprintf("result[[\"status\"]] = %d (0 indicates success)\n",
            result[["status"]]))

Delta_theta_min <- result[["solution"]][1:half_num_coeffs]
Delta_theta_max <- result[["solution"]][half_num_coeffs_p1:num_coeffs]
```

```
JC_param_lb <- theta_0 - Delta_theta_min
JC_param_ub <- theta_0 + Delta_theta_max

cat(sprintf("Est. spread for A = (%g, %g)\n", JC_param_lb[1]/MPa_to_GPa,
              JC_param_ub[1]/MPa_to_GPa))
cat(sprintf("Est. spread for B = (%g, %g)\n", JC_param_lb[2]/MPa_to_GPa,
          JC_param_ub[2]/MPa_to_GPa))
cat(sprintf("Est. spread for n = (%g, %g)\n", JC_param_lb[3], JC_param_ub[3]))
cat(sprintf("Est. spread for C = (%g, %g)\n", JC_param_lb[4], JC_param_ub[4]))
cat(sprintf("Est. spread for m = (%g, %g)\n", JC_param_lb[5], JC_param_ub[5]))
```

The output of the minimization is as follows:

```
result[["status"]] = 0 (0 indicates success)
Est. spread for A = (699.843, 699.843)
Est. spread for B = (866.224, 866.224)
Est. spread for n = (0.0719028, 0.123226)
Est. spread for C = (0.0045419, 0.00710257)
Est. spread for m = (0.874261, 1.05123)
```

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` and `jc_grad` functions that will work as objective and gradient functions for the `optim` function of R:

```
jc_for_min <- function(ABnCm, ep, l_epdot, T_s) {
  return (jc(ep, l_epdot, T_s,
             ABnCm[1], ABnCm[2], ABnCm[3], ABnCm[4], ABnCm[5]))
}

jc_for_max <- function(ABnCm, ep, l_epdot, T_s) {
  return (-jc_for_min(ABnCm, ep, l_epdot, T_s))
}

jc_grad_for_min <- function(ABnCm, ep, l_epdot, T_s) {
  return (as.vector(jc_grad(ep, l_epdot, T_s,
                            ABnCm[1], ABnCm[2], ABnCm[3],
                            ABnCm[4], ABnCm[5])))
}

jc_grad_for_max <- function(ABnCm, ep, l_epdot, T_s) {
  return (-as.vector(jc_grad(ep, l_epdot, T_s,
                             ABnCm[1], ABnCm[2], ABnCm[3],
                             ABnCm[4], ABnCm[5])))
}
```

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 RDS file (which can used to generate plots showing how much of the data are within bounds, such as those in Ramsey[3]):

```
num_data_pts_in_bounds <- 0

sigma_min <- rep(NA, num_data_pts)
sigma_max <- rep(NA, num_data_pts)

for (i in 1:num_data_pts) {

    result_min <- optim(theta_0, jc_for_min,
                        gr = jc_grad_for_min,
                        epsilon_p[i], log_ep_dot_vec[i], T_star[i],
                        method = "L-BFGS-B",
                        lower = JC_param_lb,
                        upper = JC_param_ub)

    if (result_min[["convergence"]] != 0) {
      cat(sprintf("Cannot find sigma_min for data point %s!", i))
    }

    result_max <- optim(theta_0, jc_for_max,
                        gr = jc_grad_for_max,
                        epsilon_p[i], log_ep_dot_vec[i], T_star[i],
                        method = "L-BFGS-B",
                        lower = JC_param_lb,
                        upper = JC_param_ub)

    if (result_max[["convergence"]] != 0) {
      cat(sprintf("Cannot find sigma_max for data point %s!", i))
    }

    sigma_min[i] = result_min[["value"]]
    sigma_max[i] = -result_max[["value"]]

    num_data_pts_in_bounds <- num_data_pts_in_bounds +
        as.numeric((sigma_min[i] <= sigma[i]) &&
                   (sigma[i] <= sigma_max[i]))
}

cat(sprintf("Fraction of data points in bounds = %g",
            num_data_pts_in_bounds/num_data_pts))

saveRDS(
    list(
        sigma_min = sigma_min/MPa_to_GPa,
        sigma_max = sigma_max/MPa_to_GPa
    ),
    file.path(
        "rds_data_files",
        "jc_MIDAS_rstan_IPM_sigma_bounds_strong_prior_on_A_beta0.9_farea0.75.rds"
    )
)
```

The resulting text output is as follows:

```
Fraction of data points in bounds = 0.999455
```

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.[3]

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. This is done as follows, with the contents of the file being stored in the data frame `jc_samples`:

```
jc_samples <- read.csv(
    file.path("samples",
              "jc_MIDAS_rstan_samples_weak_prior_bTQ09_fA075.csv.gz"))
```

The CSV file named previously has column headers corresponding to the names of model parameters ("A", "B", etc.), so the values in columns of this CSV file, which contain the MCMC samples for those parameters, can be accessed as jc_samples[["A"]], jc_samples[["B"]], and so on. However, the vector of samples for model parameter $SD_{\sigma,1}$ (or sd_sigma[1]) is jc_samples[["sd_sigma.1."]], with a period (“.”) replacing the opening and closing brackets in the parameter name, or alternatively, jc_samples[[make.names("sd_sigma[1]")]].

At this point, one may compute histograms that approximate the marginal PDFs of the parameters as follows:

```
jc_hists <- list()

for (param in names(jc_samples)) {
    jc_hists[[param]] <- hist(jc_samples[[param]],
                              breaks = "FD",
                              plot = FALSE)
}
```

The function call names(jc_samples) returns a vector of the names of the columns in the data frame jc_samples. The argument “breaks = "FD"” causes R to use the Freedman-Diaconis algorithm to determine the number of bins in the histogram. The argument “plot = FALSE” prevents R from actually plotting the histograms, since this is to be done in later steps. Instead, the boundaries of the bins in each histogram are stored in the vector jc_hists[[param]][["breaks"]], the number of samples in each bin is stored in the vector jc_hists[[param]][["counts"]], and jc_hists[[param]][["density"]] is the result of dividing jc_hists[[param]][["counts"]] by a normalizing factor such that the total 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 R code for calculating the prior PDFs is shown:

```
library(jsonlite)

parent_dir <- dirname(getwd())
```

```
JC_priors <- read_json(file.path(parent_dir,
                                 "Other_data", "JC_priors.json"),
                       simplifyVector = TRUE)

prior_curves <- list()

for (param in names(jc_samples)) {

    if (param == "n") {
        # Parameter "n" has a beta distribution for a prior
        prior_x <- seq(0, 1, length.out = 100)

        prior_curves[[param]] <- list(
            x = prior_x,
            y = dbeta(prior_x,
                      JC_priors[["n_alpha"]], JC_priors[["n_beta"]])
        )
    } else if (param != "lp__") {
        # All other priors have an approximately normal distribution.
        # (It's approximate because it is truncated near zero.)

        if (param == "sd_sigma.1.") {
            guess_mean <- JC_priors[["sd_sigma_guess_mean"]][1]
            guess_sd <- JC_priors[["sd_sigma_guess_sd"]][1]
        } else if (param == "sd_sigma.2.") {
            guess_mean <- JC_priors[["sd_sigma_guess_mean"]][2]
            guess_sd <- JC_priors[["sd_sigma_guess_sd"]][2]
        } else {
            guess_mean <- JC_priors[[sprintf("%s_guess_mean", param)]]
            guess_sd <- JC_priors[[sprintf("%s_guess_sd", param)]]
        }

        hist_x <- jc_hists[[param]][["breaks"]]

        prior_x_min <- min(guess_mean - 3*guess_sd, hist_x[1])
        prior_x_max <- max(guess_mean + 3*guess_sd, hist_x[length(hist_x)])

        prior_x <- seq(prior_x_min, prior_x_max, length.out = 100)

        prior_curves[[param]] <- list(
            x = prior_x,
            y = dnorm(prior_x, guess_mean, guess_sd)
        )
    }
}
```

In this code, the function `dbeta` calculates the probability density for a beta distribution. Its first argument is a vector of values for which the probability density is to be calculated, and the next two arguments are the α and β parameters of the distribution. The `dnorm` function 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 list `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 and other details of the plots' appearance). In this code, the argument "`freq = FALSE`" is passed to the `plot` function so that the counts in the histogram bins are normalized such that the area under the histogram is 1. The resulting plots are shown in Fig. 13.

```
# Setting default values for x-axis labels and legend locations
x_labels <- list()
legend_locations <- list()

for (param in names(jc_samples)) {
    x_labels[[param]] <- param
    legend_locations[[param]] <- "topright"
}

# Modifying x-axis labels
for (param in c("A", "B")) {
    x_labels[[param]] <- sprintf("%s (MPa)", param)
}

x_labels[["sd_sigma.1."]] <- expression(paste(SD[list(sigma,1)], " (MPa)"))
x_labels[["sd_sigma.2."]] <- expression(paste(SD[list(sigma,2)], " (MPa)"))

# Changing legend position of parameter "C"
legend_locations[["C"]] <- "topleft"

# Setting up line types and colors for the histogram
# and prior
hist_lty <- 0
hist_col <- "blue"

prior_lty <- 1
prior_col <- "black"

# Plotting histograms with their associated priors
for (param in names(jc_samples)) {

    if (param != "lp__") {
        prior_x <- prior_curves[[param]][["x"]]
        prior_y <- prior_curves[[param]][["y"]]

        posterior_hist <- jc_hists[[param]]

        out_pdf_name <- sprintf(
            "jc_prior_vs_marg_posterior_for_%s_weak_prior_bTQ09_fA075.pdf",
            param)
```

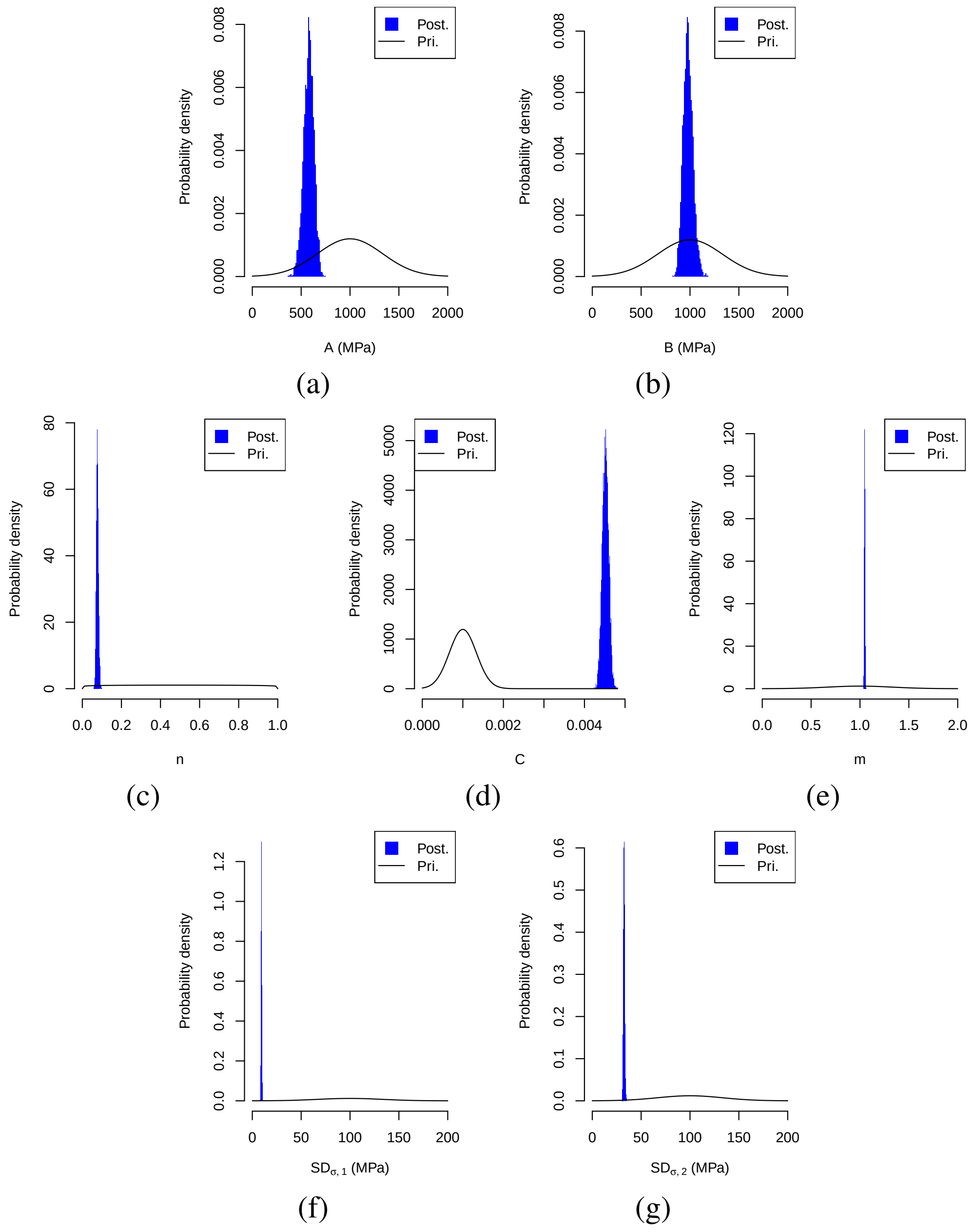


Fig. 13 Histograms approximating the posterior marginal PDFs of Johnson-Cook model parameters and nuisance parameters $SD_{\sigma,1}$ and $SD_{\sigma,2}$. These are generated from samples of an RStan MCMC run with $\beta_{TQ} = 0.9$, $f_{area} = 0.75$, and weakly informative priors. Priors are superimposed over the histograms.

```
        # Open ".pdf" file for plotting
        pdf(file = file.path("plot_files", out_pdf_name),
            title = out_pdf_name,
            pointsize = 10,
            width = 3, height = 3.5)

        # Margin adjustments
        par(mar = c(4,5,0.7,0.7), oma = c(0,0,0,0))

        plot(posterior_hist,
             freq = FALSE,
             xlim = range(c(prior_x, posterior_hist[["breaks"]])),
             ylim = range(c(prior_y, posterior_hist[["density"]])),
             xlab = x_labels[[param]],
             ylab = "Probability density",
             main = NULL,
             lty = hist_lty,
             col = hist_col)

        lines(prior_x, prior_y, lty = prior_lty, col = prior_col)

        # The use of "pch" here allow a colored box to be used
        # in the legend to indicate the histogram, while a line
        # indicates the prior.
        legend(legend_locations[[param]],
               legend = c("Post.", "Pri."),
               lty = c(hist_lty, prior_lty),
               col = c(hist_col, prior_col),
               pch = c(15, NA), pt.cex = 2)

        dev.off() # Close ".pdf" file
    }
}
```

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.[3] Similar histograms are also presented in Ramsey[3] 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 different values of β_{TQ} and f_{area} are compared. First, MCMC samples associated with these values are read in as follows. These samples are for Johnson-Cook model fits with weakly informative priors.

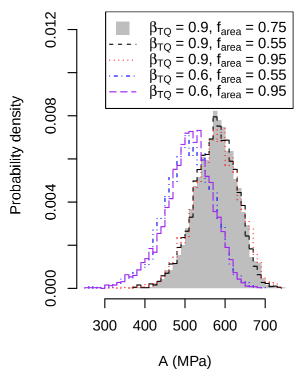
```
bTQ_fA_strs <- list(
    c("0.9", "0.75"), c("0.9", "0.55"), c("0.9", "0.95"),
    c("0.6", "0.55"), c("0.6", "0.95")
)

jc_samples <- list()

for (bTQ_fA in bTQ_fA_strs) {

    bTQ <- bTQ_fA[1]
    fA <-  bTQ_fA[2]

    # Removing the decimal points from bTQ and fA, i.e.
    # 0.9 and 0.75 become 09 and 075
    bTQ_no_decimal <- sub(".", "", bTQ, fixed = TRUE)
    fA_no_decimal <- sub(".", "", fA, fixed = TRUE)

    csv_file_name <- sprintf(
        "jc_MIDAS_rstan_samples_weak_prior_bTQ%s_fA%s.csv.gz",
        bTQ_no_decimal, fA_no_decimal)

    jc_samples[[paste(bTQ, fA, sep = ",")]] <-
        read.csv(file.path("samples", csv_file_name))
}
```

Here, `jc_samples` is a list of data frames, where each frame is associated with a pair of β_{TQ} and f_{area} values. Next, histograms are computed. Again, the argument "`breaks = "FD"`" causes R to use the Freedman-Diaconis algorithm to determine the number of bins in the histogram, while the argument "`plot = FALSE`" prevents R from actually plotting the histograms:

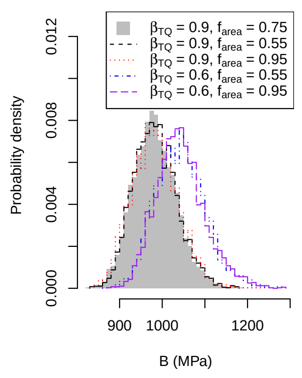
```
jc_hists <- list()

for (bTQ_fA_str in names(jc_samples)) {
    curr_jc_samples <- jc_samples[[bTQ_fA_str]]
    hist_list <- list()

    for (param in names(curr_jc_samples)) {
        if (param != "lp__") {
            hist_list[[param]] <- hist(curr_jc_samples[[param]],
                                       breaks = "FD", plot = FALSE)
        }
    }

    jc_hists[[bTQ_fA_str]] <- hist_list
}
```

Here, the histograms are stored in a list of lists named `hist_list`, where each element of the outer list is a list of histograms that has one histogram for each parameter. Each element of the outer list is associated with a pair of β_{TQ} and f_{area} values. These histograms are plotted as shown in the following R code. 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 as lines that show the outlines of the bars, via the `hist_outline` function from the R source file `bayes-stress-strain-utils.R` in Appendix C. The resulting plots are shown in Fig. 14.

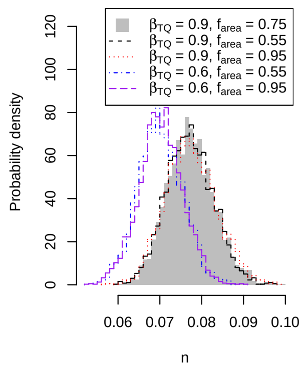
```
source("bayes-stress-strain-utils.R")

# Setting up x-axis labels, the line types and colors, and legend labels
# used in the plot.
params <- names(jc_hists[[1]])

x_labels <- list()
for (param in params) {
    x_labels[[param]] <- param
}

for (param in c("A", "B")) {
    x_labels[[param]] <- sprintf("%s (MPa)", param)
}

for (i in 1:2) {
    x_labels[[sprintf("sd_sigma.%d.",i)]] <- parse(
        text = sprintf("paste(SD[list(sigma,%d)], ' (MPa)')", i)
    )
}

line_types <- 2:5

hist_col = "gray"
line_cols <- c("black", "red", "blue", "purple")

legend_labels <- rep(NA, length(bTQ_fA_strs))
for (i in 1:length(bTQ_fA_strs)) {
    legend_labels[i] <- parse(
        text = sprintf("paste(beta[TQ], ' = %s, ', f[area], ' = %s')",
                       bTQ_fA_strs[[i]][1], bTQ_fA_strs[[i]][2]))
}

# Plotting the actual superimposed histograms
for (param in params) {
    out_pdf_name <- sprintf("jc_hists_for_%s_weak_prior.pdf",
```



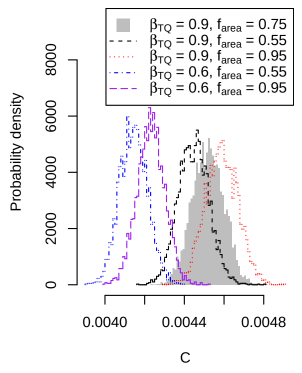
(a)



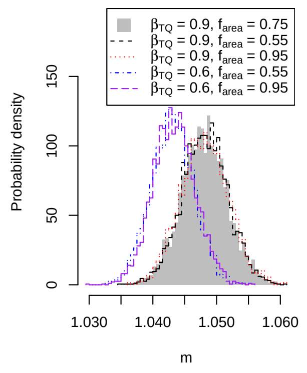
(b)



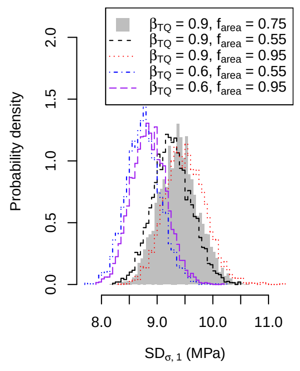
(c)



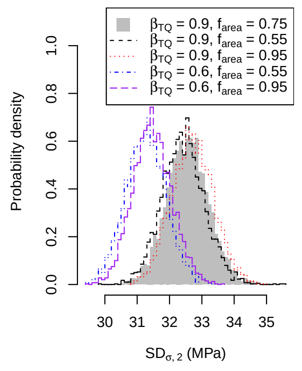
(d)



(e)



(f)



(g)

Fig. 14 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 RStan MCMC runs with the values of β_{TQ} and f_{area} in Table 1, and weakly informative priors.

```
                         param)

    pdf(file = file.path("plot_files", out_pdf_name),
        title = out_pdf_name,
        pointsize = 10,
        width = 3, height = 3.5)

    # Finding the range of x- and y-values to be shown in the plot
    xlim <- NULL
    ylim <- NULL

    for (i in 1:length(bTQ_fA_strs)) {
        curr_hist <- jc_hists[[i]][[param]]
        xlim <- range(c(xlim, range(curr_hist[["breaks"]])))
        ylim <- range(c(ylim, range(curr_hist[["density"]])))
    }

    ylim[2] <- ylim[2] + 0.5*(ylim[2] - ylim[1])

    # Margin adjustments
    par(mar = c(4,5,0.7,0.7), oma = c(0,0,0,0))

    plot(jc_hists[[1]][[param]],
         freq = FALSE,
         xlab = x_labels[[param]],
         ylab = "Probability density",
         main = NULL,
         xlim = xlim,
         ylim = ylim,
         lty = 0,
         col = hist_col
    )

    # Adding histograms to plot
    for (i in 2:length(bTQ_fA_strs)) {
        hist_outline(jc_hists[[i]][[param]],
                     lty = line_types[i-1], col = line_cols[i-1])
    }

    # Adding legend to plot
    legend("topright",
           legend = legend_labels,
           lty = c(0,line_types),
           col = c(hist_col, line_cols),
           pch = c(15, rep(NA, length(line_types))),
           pt.cex = 2,
           y.intersp = 0.9)

    dev.off() # Closing plot window
}
```

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.[3] Similar histograms are also presented in Ramsey[3] for the parameters of the Zerilli-Armstrong (BCC) model (and associated nuisance parameters), for both the fits to all available MIDAS data and the fits to only 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 in Section 7.2, into a list of data frames named `jc_samples`, where each frame is associated with a string 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 RDS files (as discussed in Section 6.2), and these are loaded again as shown, with `bTQ_fA_strs` defined as in Section 7.2:

```
main_data <- readRDS(file.path("rds_data_files",
                               "Main_data_for_JC.rds"))

temp_data <- list()
for (bTQ_fA in bTQ_fA_strs) {
    bTQ <- bTQ_fA[1]
    fA <-  bTQ_fA[2]

    temp_data[[paste(bTQ, fA, sep = ",")]] <- readRDS(
        file.path("rds_data_files",
                  sprintf("T_beta%s_farea%s_JC.rds", bTQ, fA)))
}
```

Again, the function `expose_stan_functions` from RStan is used to load a function `jc` representing the Johnson-Cook model into an R session. However, 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 their corresponding Stan models. In contrast to the previous instance of `sigma_model_func` from Section 5.2, where the strain is a vector and the model parameters are scalars, here the strain is a scalar, and the MCMC samples for each model parameter are passed as vectors:

```
library(rstan)
jc_model <- readRDS(file.path("compiled_stan_models", "jc.rds"))
expose_stan_functions(jc_model)
```

```r
sigma_model_func <- function(epsilon_p,
                             epsilon_p_dot,
                             temperature,
                             theta_model) {
    log_epsilon_p_dot <-
        log(epsilon_p_dot/theta_model[["epsilon_p_dot_0"]])

    T_star <- (temperature - theta_model[["T_room"]])/
      (theta_model[["T_melt"]] - theta_model[["T_room"]])

    A <- theta_model[["A"]]
    B <- theta_model[["B"]]
    n <- theta_model[["n"]]
    C <- theta_model[["C"]]
    m <- theta_model[["m"]]

    sigma_samples <- numeric(length(A))

    # A "for" loop is needed because expose_stan_functions
    # generates the "jc" function in such a way that it cannot
    # accept vectors for the model parameter arguments A, B, n,
    # C, and m.
    for (i in 1:length(A)) {
        sigma_samples[i] <- jc(epsilon_p, log_epsilon_p_dot,
                               T_star,
                               A[i], B[i], n[i], C[i], m[i])
    }

    return (sigma_samples)
}
```

Since the MCMC samples differ for different values of β_{TQ} and f_{area}, different values of `theta_model` are needed for those values, so a list of the needed values is created. A list of samples of $SD_{\sigma,1}$ and $SD_{\sigma,2}$ is generated as well for those values of β_{TQ} and f_{area}:

```r
theta_model_list <- list()
sd_sigma_list <- list()

for (bTQ_fA in bTQ_fA_strs) {
    bTQ <- bTQ_fA[1]
    fA <-  bTQ_fA[2]
    bTQ_fA_str <- paste(bTQ, fA, sep = ",")

    curr_samples <- jc_samples[[bTQ_fA_str]]

    theta_model_list[[bTQ_fA_str]] <- list(
        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_list[[bTQ_fA_str]] <- list(
        curr_samples[["sd_sigma.1."]],
        curr_samples[["sd_sigma.2."]]
    )
}
```

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`.[9] 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.[11] The statistics of the PPDs are stored in lists of vectors `ppd_mean`, `ppd_hdi_min`, and `ppd_hdi_max`. For each PFP, the bounds of the 95% HDI are computed and stored in the lists of vectors `pfp_hdi_min` and `pfp_hdi_max`. These lists of vectors are created via the following R code:

```
library(HDInterval)

ppd_mean <- list()
ppd_hdi_min <- list()
ppd_hdi_max <- list()

pfp_hdi_min <- list()
pfp_hdi_max <- list()

num_curves <- main_data[["num_curves"]]
curve_sizes <- main_data[["curve_sizes"]]
epsilon_p <- main_data[["epsilon_p"]]
epsilon_p_dot <- main_data[["epsilon_p_dot"]]

for (bTQ_fA in bTQ_fA_strs) {
    bTQ <- bTQ_fA[1]
    fA <-  bTQ_fA[2]

    bTQ_fA_str <- paste(bTQ, fA, sep = ",")

    temperature <- temp_data[[bTQ_fA_str]]
    theta_model <- theta_model_list[[bTQ_fA_str]]
    sd_sigma <- sd_sigma_list[[bTQ_fA_str]]

    curr_ppd_mean <- numeric(sum(curve_sizes))
    curr_ppd_hdi_min <- numeric(sum(curve_sizes))
    curr_ppd_hdi_max <- numeric(sum(curve_sizes))

    curr_pfp_hdi_min <- numeric(sum(curve_sizes))
    curr_pfp_hdi_max <- numeric(sum(curve_sizes))

    start_ind <- 1
    for (curve_ind in 1:num_curves) {
        end_ind <- start_ind + curve_sizes[curve_ind] - 1

        if (epsilon_p_dot[curve_ind] <= 1.0) {
            curr_sd_sigma <- sd_sigma[[1]]
        } else {
            curr_sd_sigma <- sd_sigma[[2]]
        }

        for (i in start_ind:end_ind) {
            curr_pfp <-
                sigma_model_func(epsilon_p[i],
                                 epsilon_p_dot[curve_ind],
                                 temperature[i],
                                 theta_model)

            curr_ppd <- rnorm(length(curr_pfp),
                              curr_pfp, curr_sd_sigma)

            curr_ppd_mean[i] <- mean(curr_ppd)

            # By default, the hdi function computes the bounds
            # of the 95% HDI.
            curr_ppd_hdi_range <- hdi(curr_ppd)
            curr_ppd_hdi_min[i] <- curr_ppd_hdi_range[1]
            curr_ppd_hdi_max[i] <- curr_ppd_hdi_range[2]

            curr_pfp_hdi_range <- hdi(curr_pfp)
            curr_pfp_hdi_min[i] <- curr_pfp_hdi_range[1]
            curr_pfp_hdi_max[i] <- curr_pfp_hdi_range[2]
        }

        start_ind <- end_ind + 1
    }

    ppd_mean[[bTQ_fA_str]] <- curr_ppd_mean
    ppd_hdi_min[[bTQ_fA_str]] <- curr_ppd_hdi_min
    ppd_hdi_max[[bTQ_fA_str]] <- curr_ppd_hdi_max

    pfp_hdi_min[[bTQ_fA_str]] <- curr_pfp_hdi_min
    pfp_hdi_max[[bTQ_fA_str]] <- curr_pfp_hdi_max
}
```

The variable curr_ppd shown previously is a vector of samples of the PPD associated with the strain and temperature values epsilon_p[i] and temperature[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 qth MCMC sample of the model parameters. Each sample curr_ppd[q] of the PPD is also drawn from a normal distribution. Here, the function rnorm performs such draws. Its first argument is the number of draws, and the remaining arguments are such that draw q comes from a distribution with mean curr_pfp[q] and standard deviation curr_sd_sigma[q]. Element curr_pfp[q] is, of course, a sample of the PFP determined from the qth MCMC sample of the model parameters. Given the previous code, provided that $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$, the R expression ppd_mean[["0.9,0.75"]][i] indicates the mean of the PPD of where the plastic strain is epsilon_p[i], the temperature is temperature[i], and the plastic strain rate is epsilon_p_dot[curve_ind], where curve_ind determines the range of values for i. The R expressions ppd_hdi_min[["0.9,0.75"]][i] and ppd_hdi_max[["0.9,0.75"]][i] indicate the bounds of the 95% HDI of that PPD. The hdi function from the R package HDInterval[*] is used to estimate the HDIs.

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

```r
source("bayes-stress-strain-utils.R")

T_init_str <- c("298", "298",  "298",  "298", "473",  "673",  "873")
sigma <- main_data[["sigma"]]

# Setting up line types and colors

# For beta_TQ = 0.9, f_area = 0.75
line_type_mean_only <- 1
col_val_hdi_only <- "skyblue"
col_val_mean_only <- "purple"

# For other beta_TQ and f_area pairs
line_types_mean_and_hdi <- 2:5
col_vals_mean_and_hdi <- c("red", "brown", "green", "blue")
```

[*]If the advice in Section 2 has been followed, then this package should already have been installed.

```
# For experimental data
col_val_data <- "black"

# Setting up legend labels
bTQ_fA_legend_paste_strs <- sapply(bTQ_fA_strs,
                                   function(bTQ_fA) {
    sprintf("beta[TQ], ' = %s, ', f[area], ' = %s'",
            bTQ_fA[1], bTQ_fA[2])
})

legend_labels <- c(
    parse(text = sprintf("paste('95%% HDI, ', %s)",
                         bTQ_fA_legend_paste_strs[1])),
    parse(text = sprintf("paste('Mean, ', %s)",
                         bTQ_fA_legend_paste_strs[1])),
    sapply(bTQ_fA_legend_paste_strs[2:length(bTQ_fA_legend_paste_strs)],
           function(bTQ_fA_lgd_str) {
               parse(text = sprintf("paste('Mean & 95%% HDI, ', %s)",
                                    bTQ_fA_lgd_str))
    }),
    "Exp. Data"
)

# No extra space needed for the legend for low strain rates, only
# for plots with high-strain-rate data
space_for_legend <- c(rep(0.0, times = 2), rep(0.5, times = 5))

# Plotting HDI of PPDs
start_ind <- 1
for (curve_ind in 1:num_curves) {
    end_ind <- start_ind + curve_sizes[curve_ind] - 1

    out_pdf_name <- sprintf("jc_hdi_edot%g_T%s_weak_prior.pdf",
                            epsilon_p_dot[curve_ind],
                            T_init_str[curve_ind])

    pdf(file = file.path("plot_files", out_pdf_name),
        title = out_pdf_name,
        pointsize = 10,
        width = 4, height = 4)

    curr_epsilon_p <- epsilon_p[start_ind:end_ind]
    curr_sigma <- sigma[start_ind:end_ind]

    curr_hdi_min <- list()
    curr_hdi_max <- list()
    for (i in 1:length(bTQ_fA_strs)) {
        curr_hdi_min[[i]] <- ppd_hdi_min[[i]][start_ind:end_ind]
        curr_hdi_max[[i]] <- ppd_hdi_max[[i]][start_ind:end_ind]
    }

    ymin <- min(unlist(curr_hdi_min), curr_sigma)
    ymax <- max(unlist(curr_hdi_max), curr_sigma)
```

```
ymin <- ymin - space_for_legend[curve_ind]*(ymax - ymin)

par(mar = c(4,4,2,0.7), oma = c(0,0,0,0))

make_empty_xy_plot(range(curr_epsilon_p), c(ymin, ymax))

# Plot 95% HDI for beta_TQ = 0.9, f_area = 0.75 as shaded region
fill_between_curves(curr_epsilon_p,
                    curr_hdi_min[[1]], curr_hdi_max[[1]],
                    col = col_val_hdi_only)

# Plot mean for beta_TQ = 0.9, f_area = 0.75 as lines
lines(curr_epsilon_p, ppd_mean[[1]][start_ind:end_ind],
      lty = line_type_mean_only,
      col = col_val_mean_only)

# Plot mean and bounds of 95% HDI as lines
for (i in 2:length(bTQ_fA_strs)) {
    lines(curr_epsilon_p, curr_hdi_min[[i]],
          lty = line_types_mean_and_hdi[i-1],
          col = col_vals_mean_and_hdi[i-1])

    lines(curr_epsilon_p, ppd_mean[[i]][start_ind:end_ind],
          lty = line_types_mean_and_hdi[i-1],
          col = col_vals_mean_and_hdi[i-1])

    lines(curr_epsilon_p, curr_hdi_max[[i]],
          lty = line_types_mean_and_hdi[i-1],
          col = col_vals_mean_and_hdi[i-1])
}

# Plot experimental data
pt_type_data <- 46
points(curr_epsilon_p, curr_sigma,
       col = col_val_data,
       pch = pt_type_data)

title(xlab = expression(epsilon[p]),
      ylab = expression(paste(sigma, " (MPa)")),
      main = sprintf("%s K, %g/s",
                     T_init_str[curve_ind],
                     epsilon_p_dot[curve_ind]))

legend("bottomright",
       legend = legend_labels,
       lty = c(0,
               line_type_mean_only,
               line_types_mean_and_hdi,
               0),
       col = c(col_val_hdi_only,
               col_val_mean_only,
               col_vals_mean_and_hdi,
               col_val_data),
       pch = c(15,
                   rep(NA, 1 + length(line_types_mean_and_hdi)),
                   pt_type_data),
           pt.cex = 2,
           y.intersp = 0.9)

    dev.off()

    start_ind <- end_ind + 1
}
```

This code employs two functions from the R source file `bayes-stress-strain-utils.R`. One of these is `make_empty_xy_plot`, which creates an empty plot window with given ranges for the x- and y-coordinates. The other is `fill_between_curves`, which plots a shaded region between two curves. The resulting plots are shown in Figs. 15 and 16. Plots showing the 95% HDI bounds of the PFPs are shown in Figs. 17 and 18. 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 MIDAS data and the MIDAS data used to fit the Johnson-Cook model. These are shown in Ramsey.[3]

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.[32] Both of these are fairly simple to calculate in R. For the Johnson-Cook model with weakly informative priors, $\beta_{TQ} = 0.9$, and $f_{area} = 0.75$, the correlation matrix may be evaluated as follows:

```
jc_samples <- read.csv(
    file.path("samples",
              "jc_MIDAS_rstan_samples_weak_prior_bTQ09_fA075.csv.gz"))

corr_mat_jc <- cor(jc_samples)
print(corr_mat_jc)
```

The following is the printed matrix:

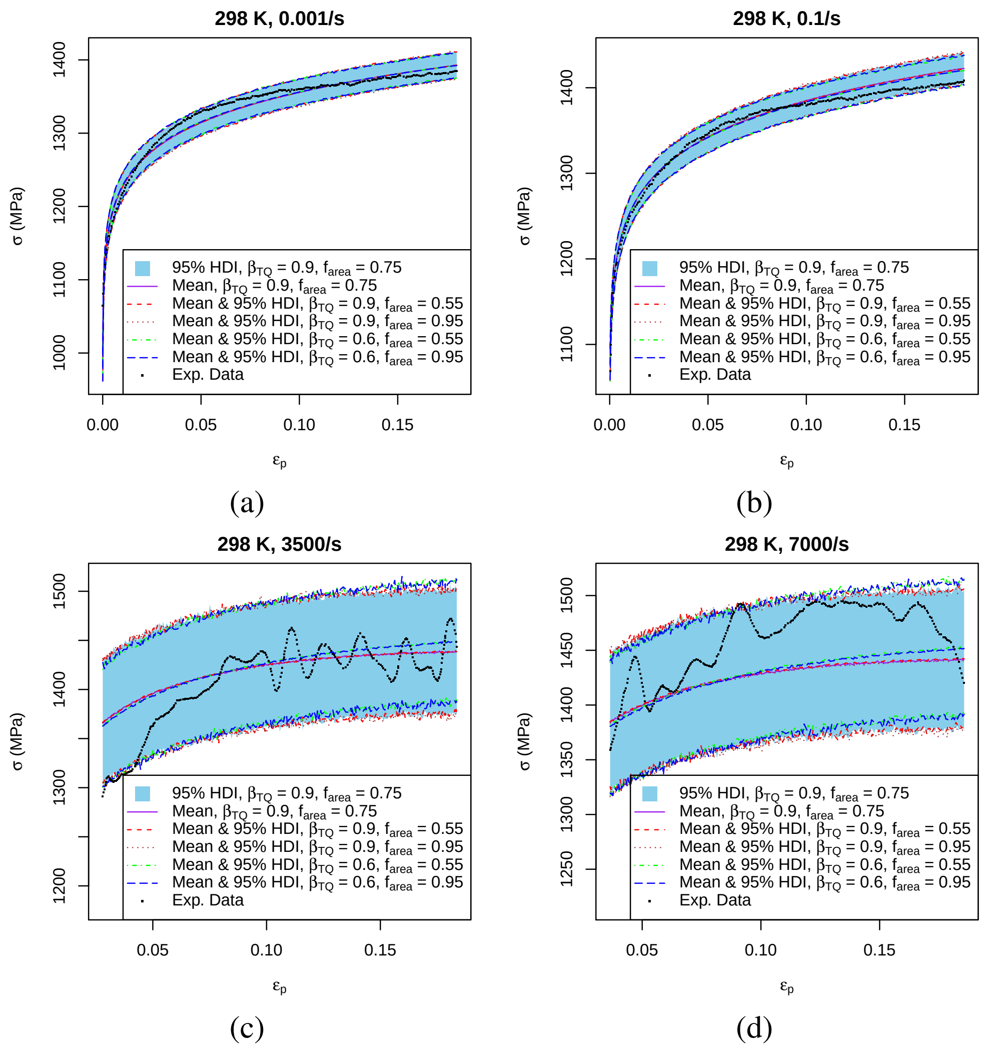
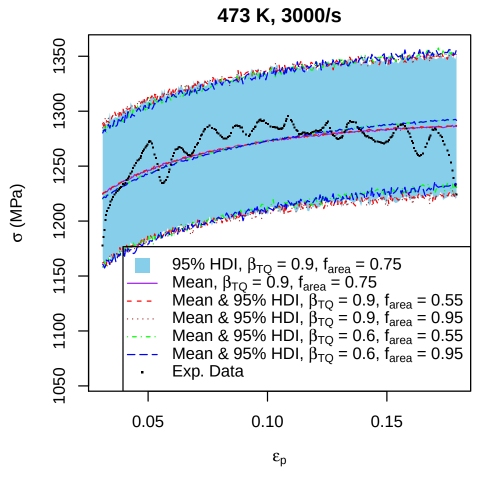
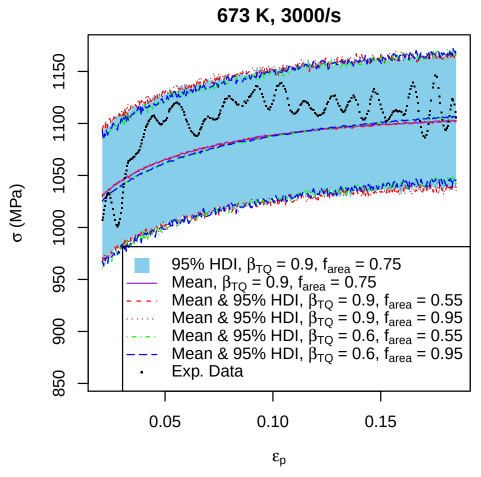


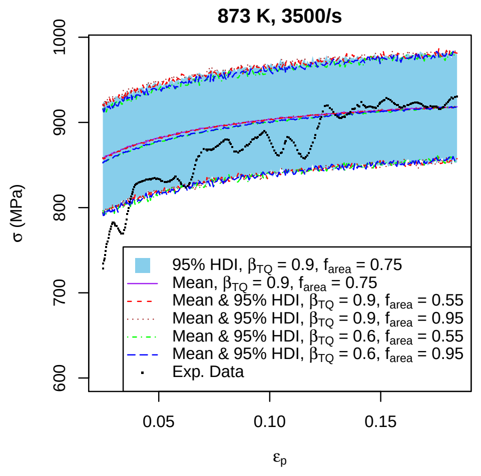
Fig. 15 Stress-strain data for initial sample temperatures of 298 K, along with estimates of the mean and the 95% HDI for PPDs generated from samples of RStan MCMC runs for the Johnson-Cook model with weakly informative priors. The 95% HDI for $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ is plotted as a shaded region between the minimum and maximum of the HDI.



(a)



(b)



(c)

Fig. 16 Stress-strain data for high initial sample temperatures along with estimates of the mean and the 95% HDI for PPDs generated from samples of RStan 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.

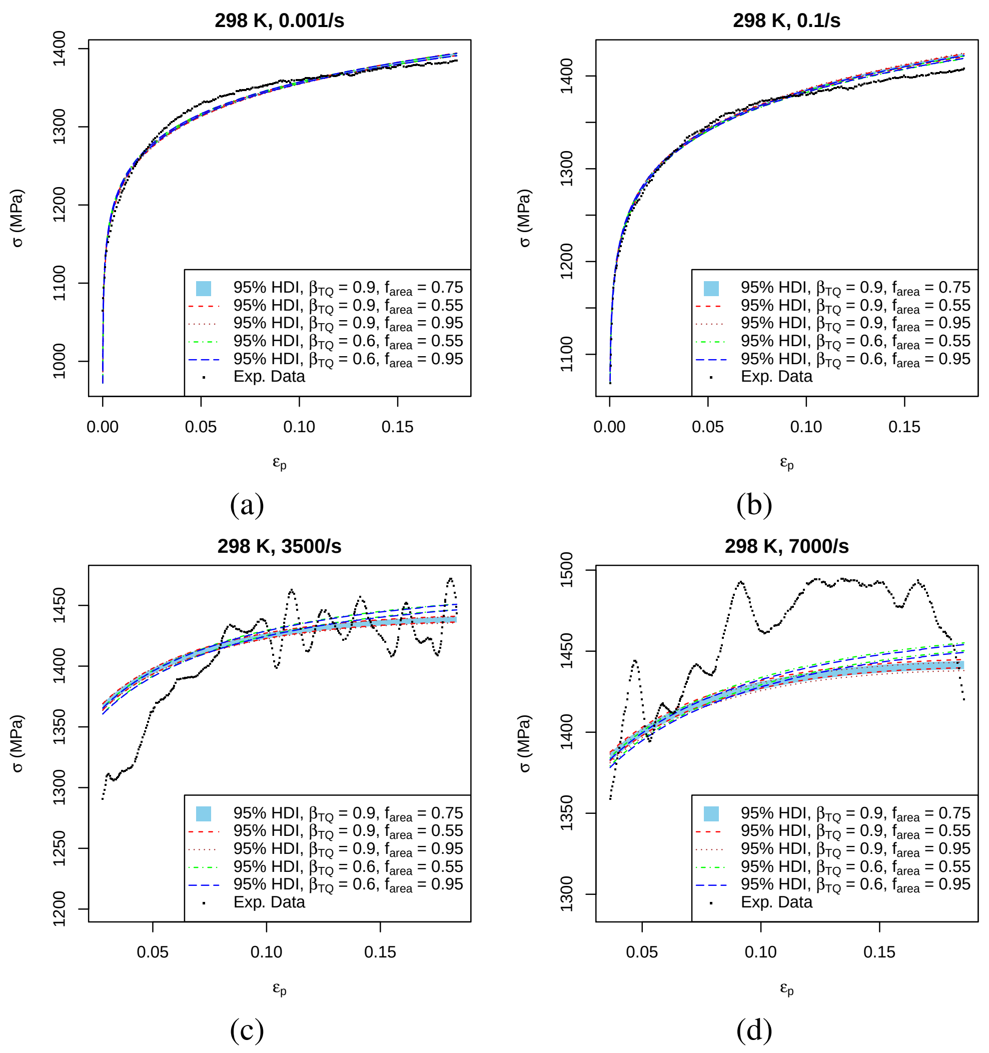
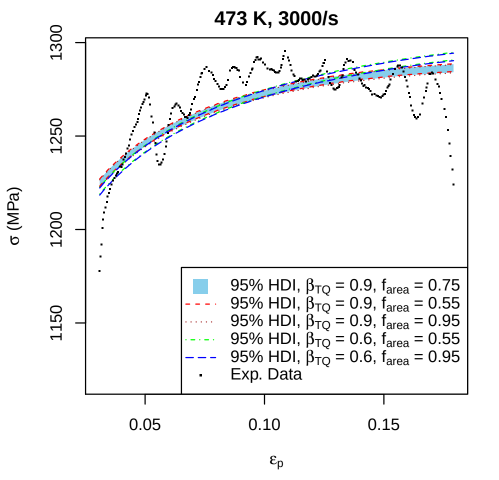
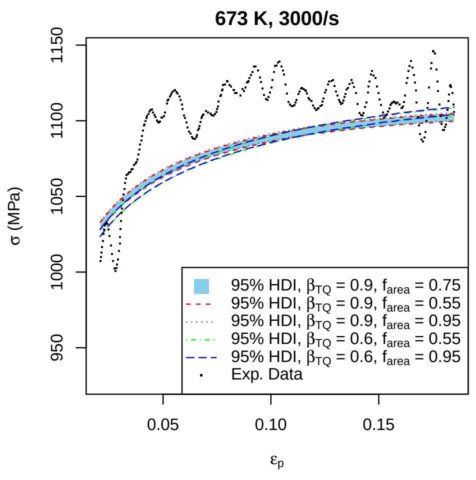


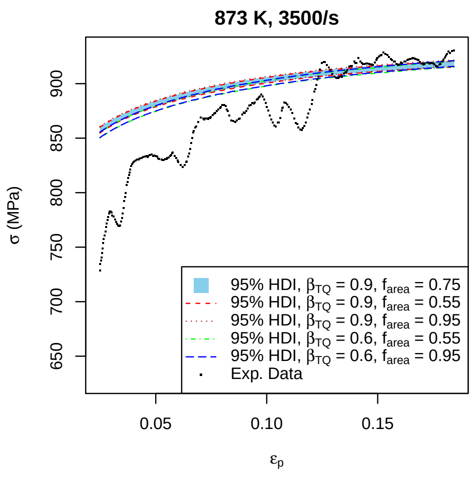
Fig. 17 Stress-strain data for initial sample temperatures of 298 K, along with estimates of the 95% HDI for PFPs generated from samples of RStan MCMC runs for the Johnson-Cook model with weakly informative priors. The 95% HDI for $\beta_{TQ} = 0.9$ and $f_{area} = 0.75$ is plotted as a shaded region between the minimum and maximum of the HDI.



(a)



(b)



(c)

Fig. 18 Stress-strain data for high initial sample temperatures along with estimates of the 95% HDI for PFPs generated from samples of RStan 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.

```
                        A           B           n           C           m
A             1.00000000 -0.99942647  0.98978369  0.06072992 -0.05456907
B            -0.99942647  1.00000000 -0.98561331 -0.04984533  0.04182829
n             0.98978369 -0.98561331  1.00000000  0.06632394 -0.06734621
C             0.06072992 -0.04984533  0.06632394  1.00000000 -0.68925783
m            -0.05456907  0.04182829 -0.06734621 -0.68925783  1.00000000
sd_sigma.1.   0.21133969 -0.19916422  0.24040565  0.29599499 -0.25753647
sd_sigma.2.  -0.11503380  0.10758398 -0.12803428 -0.25490191  0.21244271
lp__          0.07519178 -0.07739372  0.04003585 -0.01119199  0.02399214
             sd_sigma.1. sd_sigma.2.        lp__
A              0.2113397 -0.11503380  0.07519178
B             -0.1991642  0.10758398 -0.07739372
n              0.2404056 -0.12803428  0.04003585
C              0.2959950 -0.25490191 -0.01119199
m             -0.2575365  0.21244271  0.02399214
sd_sigma.1.    1.0000000 -0.13027907 -0.10100763
sd_sigma.2.   -0.1302791  1.00000000 -0.01832615
lp__          -0.1010076 -0.01832615  1.00000000
```

The function `cor` calculates the correlation coefficient for each pair of columns in the data frame `jc_samples`, and it 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 rank correlation matrix is similarly trivial:

```
rcorr_mat_jc <- cor(jc_samples, method = "spearman")
print(rcorr_mat_jc)
```

The printed rank corrrelation matrix is as follows:

```
                        A           B           n           C           m
A             1.00000000 -0.99935909  0.99420649  0.06798517 -0.06174925
B            -0.99935909  1.00000000 -0.99021729 -0.05803258  0.05000496
n             0.99420649 -0.99021729  1.00000000  0.06811196 -0.06898967
C             0.06798517 -0.05803258  0.06811196  1.00000000 -0.67384094
m            -0.06174925  0.05000496 -0.06898967 -0.67384094  1.00000000
sd_sigma.1.   0.20380761 -0.19250742  0.22506036  0.28823035 -0.25392727
sd_sigma.2.  -0.10558684  0.09837810 -0.11759043 -0.24229028  0.20593518
lp__          0.03728332 -0.03789549  0.03514480 -0.01715314  0.02674573
             sd_sigma.1. sd_sigma.2.        lp__
A             0.20380761 -0.10558684  0.03728332
B            -0.19250742  0.09837810 -0.03789549
n             0.22506036 -0.11759043  0.03514480
C             0.28823035 -0.24229028 -0.01715314
m            -0.25392727  0.20593518  0.02674573
```

```
sd_sigma.1.  1.00000000 -0.11990047 -0.09483783
sd_sigma.2. -0.11990047  1.00000000 -0.01039431
lp__        -0.09483783 -0.01039431  1.00000000
```

Again, the function `cor` is used. However, when the argument "`method = "spearman"`" is used, for each element of a column in `jc_samples`, it assigns 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.[1]

Both the correlation matrix `corr_mat_jc` and the rank correlation matrix `rcorr_mat_jc` may be saved to CSV files using the `write.csv` function.

8. Conclusions

This report describes a workflow, based on RStan, lpSolve, and the R 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 RStan, noting what to do to address them;
- estimating bounds on model parameters via an approximate IPM approach;
- generating samples of a PPD or PFP, noting how to plot 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.[32]

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. R Core Team. R: a language and environment for statistical computing—reference index. c2018 [accessed 2018 May]. https://cran.r-project.org/doc/manuals/r-release/fullrefman.pdf.

2. Guo J, Gabry J, Goodrich B, Lee D, Sakrejda K, Trustees of Columbia University, Sklyar O, The R Core Team, Oehlschlaegel-Akiyoshi J, Wickham H, de Guzman J, Fletcher J, Heller T, Niebler E. Package "rstan". c2018 [accessed 2018 Mar]. https://cran.r-project.org/web/packages/rstan/rstan.pdf.

3. 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.

4. 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.

5. 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.

6. 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.

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

8. 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.

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

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

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

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

13. 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).

14. 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.

15. 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.

16. Goodrich B. RStan: Getting started. c2018 [accessed 2018 Mar]. `https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started`.

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

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

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

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

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

22. RStudio. RStudio: Open source and enterprise-ready professional software for R. c2018 [accessed 2018 Mar]. https://www.rstudio.com/.

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

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

25. Ramsey JJ. Quantifying uncertainties in parameterizations of strength models of rolled homogeneous armor: part 3, Python-based workflow. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2019 Sep. Report No.: ARL-TR-8828.

26. 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.

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

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

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

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

31. Berkelaar M et al. lpSolve: Interface to 'Lp_solve' v. 5.5 to solve linear/integer programs. 2015 [accessed 2019 Mar]. https://cran.r-project.org/package=lpSolve.

32. 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.

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 $(\mathrm{J/kg \cdot K})$	Temp. (K)	Spec. heat $(\mathrm{J/kg \cdot K})$	Temp. (K)	Spec. heat $(\mathrm{J/kg \cdot K})$	Temp. (K)	Spec. heat $(\mathrm{J/kg \cdot K})$	Temp. (K)	Spec. heat $(\mathrm{J/kg \cdot K})$
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 R

R is the name of a software suite used for statistics and data analysis, as well as the name of the scripting language that underlies that suite.[1] A few of its key features are discussed.

B.1 Basic Syntax

The basic arithmetic operators used in R 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.

Identifiers in R, such as variables and function names, consist of a combination of letters, numbers, underscores ("`_`"), and—unlike most programming languages—also *periods* ("`.`"). Identifiers, though, cannot start with a number or underscore, nor can they be certain keywords in R, such as `if`, `while`, and so on.

Numbers in R are represented mostly straightforwardly, such that, for example, `2.3` represents the number 2.3, and either `1` or `1.0` represents the number 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 simply as `1.3 + 2.1i`. (However, the imaginary unit i is represented as `1i`, not `i`. The latter is just a variable name.) Whereas some other programming languages would treat `1` or `1.0` differently, with the former being treated as an integer and the latter as a floating-point number,[2] both are treated as floating-point numbers in R. If one must indicate an integer literal specifically, one can use the suffix `L` (e.g., `1L` for the integer 1); this is seldom needed, though.

Other literal quantities in R are character strings and Boolean values. Character strings in R are delimited by either single or double quotes. For example, either `'a`

[1] R Foundation. R: The R project for statistical computing. c2018 [accessed 2018 May]. `https://www.r-project.org/`.

[2] In many programming languages, real numbers are represented approximately via floating-point numbers; see Goldberg D. What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 1991;23(1):5–48.

string' or "a string" represents a string consisting of the characters “a”, a space, “s”, “t”, “r”, “i”, “n”, and “g”. Certain sequences in strings that begin with a backslash (“\”) are interpreted specially. In particular, "\n" indicates a newline, and "\\" indicates a literal backslash. There are two Boolean values, TRUE or FALSE, which are keywords in R. R also predefines two variables T and F, which by default have the values TRUE and FALSE, respectively. However, unlike the keywords TRUE and FALSE, variables T and F can be set to other values, though doing so may cause confusion in practice.

R 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 R language interpreter. Another is the operator “<-”, which is used to assign values to variables, much in the same way that other programming languages use “=”. The following are examples of assignment statements, with comments indicating what the assignment statement is doing:

```
x <- 1          # Assigning 1 to the variable x
y <- 2e5        # Assigning 200000 to the variable y
z <- 6.3e-2     # Assigning 0.063 to the variable z
```

In general, statements in R 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. However, the R interpreter does not treat the previous statement as an error. Rather it looks to subsequent lines to try to complete the statement. Accordingly, the following R code is correct:

```
a <- x + y +
  z
```

The following is also correct:

```
a <-
x + y + z
```

Relational and logical operators are usually used with the control structures discussed in Section B.7. 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. The logical operator "!" indicates negation, so that !TRUE is FALSE and vice versa. The logical operators "&" and "&&" both represent a Boolean "and" operation (i.e., x & y is TRUE only if both x and y are TRUE), but they are not entirely synonymous. Unlike the "&" operator, "&&" short-circuits; if its first operand evaluates to FALSE, the second operand is never evaluated. Short-circuiting, though, does not work with vectors or arrays (data structures discussed in more detail in Sections B.5.1 and B.5.3). The logical operators "|" and "||", which both represent a Boolean "or" operation (i.e., x | y is FALSE only if both x and y are FALSE), work similarly. The operator "||" short-circuits (only evaluating its second operand if its first operand is FALSE), and "|" is to be used with vectors and arrays.

B.2 Format Strings with `sprintf`

Format strings are a kind of template. They contain placeholders that begin with "%" and end with a character, such as "%s", which can be substituted by other expressions. An example format string would be "Real part of z = %g, imaginary part of z = %g". To use a format string, one uses the sprintf function, as shown in the following example code:

```
z <- 2e6 + 3.2i
out_string <- sprintf("Real part of z = %g, imaginary part of z = % g\n",
                      Re(z), Im(z))

cat(out_string)
```

The function sprintf creates the string "Real part of z = 2e+06, imaginary part of z = 3.2\n." This string is then assigned to the variable out_string. and the function cat prints out_string. The string "%g" is a placeholder for a real number that may cause the number to be displayed in scientific notation (using the "1.23e4" syntax discussed in Section B.1) if it is large enough. Other placeholders include "%d", for integers, and "%s", for strings. The function sprintf replaces the doubled percent sign ("%%") in a format string with a literal "%".

B.3 Function Definition and Invocation

In R, functions take arguments, perform some sequence of operations with those arguments, and return a value. In some case, such as with functions that plot graphs or write to files, the return value is immaterial and usually ignored, but is technically always there. R has several built-in functions, and users can define their own functions as well. A somewhat contrived example of this is shown:

```
qd_formula <- function(a, b = 0, c = 0) {
   sqrt_b2_minus_4ac <- sqrt(as.complex(b^2 - 4*a*c))
   two_a <- 2*a

   x <- c((-b - sqrt_b2_minus_4ac)/two_a,
          (-b + sqrt_b2_minus_4ac)/two_a)

   return(x)
}
```

This creates a function named `qd_formula`, which implements the quadratic formula $x = (-b \pm \sqrt{b^2 - 4ac})/2a$. The built-in R function `sqrt` implements the square root. The function `as.complex` ensures that the value of its argument is treated as a complex number, so that `sqrt` returns an imaginary value if `b^2 - 4*a*c` is negative. The variable `x` is set to a vector whose two elements are the two possible values of the quadratic formula. (More about vectors and the function `c` that creates them is discussed in Section B.5.) Finally, the value of `x` is returned by the function. Strictly speaking, a `return` statement is not needed. The function could have been written as

```
qd_formula <- function(a, b = 0, c = 0) {
   sqrt_b2_minus_4ac <- sqrt(as.complex(b^2 - 4*a*c))
   two_a <- 2*a

   c((-b - sqrt_b2_minus_4ac)/two_a,
     (-b + sqrt_b2_minus_4ac)/two_a)
}
```

and the last line of the function would have caused the same values to be returned. However, for functions that consist of more than a single statement, a `return` statement is usually more readable.

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*. 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. Several built-in R functions 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.

B.4 Cross-platform File Path Functions

R 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`. Now, technically, on Windows, `path/to/file` works as well, at least for use in R. However, to ensure portability to any platform, it is best to specify the file path in R as `file.path("path", "to", "file")`.

There are a few other R functions that pertain to file paths. One of them is `getwd`, which takes no arguments and returns the current working directory as an absolute path. Any relative paths used in R are interpreted with respect to this directory, so that, for example, the path `file.path("path", "to", "file")` is taken to be within this directory. Two other functions are `basename` and `dirname`. The first function returns the final component of a path, so `basename("path/to/file")` returns `"file"`. The other function returns the directory that contains the final component of the path, so `dirname("path/to/file")` returns `"path/to"`. These functions can be used, for example, to obtain the parent of the current working directory as follows:

```r
parent_dir <- dirname(getwd())
```

B.5 Data Structures

In practical cases, variables in R often refer to more than just ordinary numbers, and are often used to store sometimes complicated data structures. Some of the common data structures are discussed in this section.

B.5.1 Vectors

The simplest data structure in R is a vector, which is a sequence of values. For example, a vector named `x`, with the elements 130.7, 4, 3×10^2, 7.1, 2.4×10^3, and 1.2×10^{-5}, may be created as follows:

```r
x <- c(130.7, 4, 3e2, 7.1, 2.4e3, 1.2e-5)
```

One can loosely describe `c(x1,x2,...)` as a notation for expressing a vector that has the elements `x1`, `x2`, and so on. However, technically what is happening here is that the function `c` is returning a value that is a concatenation of all its arguments, where the type of value it returns depends on the type of its arguments. In the previous R code, the arguments of `c` are actually vectors—since in R, numbers are just numeric vectors with only one element—so what the function `c` returns is another vector. In the following code,

```r
y <- c(x, c(5,4))
```

the function `c` creates a new vector (assigned to `y`) whose elements are 130.7, 4, 3×10^2, 7.1, 2.4×10^3, 1.2×10^{-5}, 5, and 4, where all but the last two elements are the same as the elements of `x`.

To access parts of a vector, one may use the "`[...]`" operator. For example, `x[3]` is the third element of `x`, 3×10^2. Here, the value of 3 between the "`[`" and "`]`" is called an *index*. The expression `x[2:4]` is a vector containing the second through the fourth elements, that is, 4, 3×10^2, and 7.1. Vectors can be accessed by more than just numeric indices. For example, if one executes the following R statement,

```r
names(x) <- c("one", "two", "fish", "A", "B", "C")
```

then one can use `x["fish"]` access the third element of `x`. The previous R statement also showcases an instance where the elements of a vector are character strings rather than numbers. In general, the only restriction on the types of elements in a vector is that they be all of the same type.

Also, if one creates the following vector of Boolean values

```
logical_inds <- c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE)
```

then x[logical_inds] is a vector consisting of the first, second, and last elements x, that is, the elements of x that correspond to the elements of logical_inds that are TRUE. This is called logical indexing. Typically, logical indices are created with relational and/or logical operators. For example, if logical_inds2 is defined such that

```
logical_inds2 <- (x > 100) & (x < 1000)
```

then logical_inds2 is the vector c(TRUE, FALSE, TRUE, FALSE, FALSE, FALSE), and x[logical_inds2] returns a vector of the elements of x that are greater than 100 and less than 1000. When defining logical indices, the short-circuiting logical operators ("&&" and "||") tend to produce wrong results. For example, if the operator "&&" were used instead of "&" in the previous R code, then logical_inds2 would just be the value TRUE, and x[logical_inds2] would return *all* the elements of x, which is unlikely to be what one intended.

Arithmetic operations done on vectors typically operate elementwise, so for example, c(a,b)*c(d,e) is the same as c(a*d, b*e). They also work in cases where one operand is a vector and the other operand is an ordinary number, so that, for example c(a,b) + d is c(a+d, b+d), and c(a,b)^d is c(a^d, b^d). These arithmetic operators even return values in cases where the vector operands are of different sizes, although the results may not be mathematically useful. The logical operators "&" and "|" also operate elementwise on vectors. Functions that are built into R often operate elementwise on vectors as well. For example, log(c(a,b)) is equivalent to c(log(a), log(b)).

There are a few functions that are used to get properties of vectors. One of them, the names, has been shown before. In the use shown previously, one adds labels to a vector x by assigning to names(x). However, in the reverse case, where

```
names_x <- names(x)
```

the function names is used to retrieve the labels of x and store them (as a vector of strings) in the variable names_x. Another function, and one far more commonly used, is `length`, which returns the number of elements in a vector.

There are also functions and expressions that can be used to create certain kinds of vectors. For example, in the expression `seq(x_start, x_end, length.out = 100)`, the function `seq` creates a vector with 100 elements, where the first element is `x_start`, the last element is `x_end`, and the rest of the elements are evenly spaced between those two values, that is, the difference between successive elements is `(x_end - x_start)/(100 - 1)`. The expression `seq(1,10,2)` creates a sequence of integers that starts at 1 and continues in increments of 2 until it reaches the largest possible number that is no greater than 10 (i.e., the odd numbers 1, 3, 5, 7, and 9). The expression `i_start:i_end`, where `i_start` and `i_end` are integer values, is the sequence of integers from `i_start` through `i_end`. For example, `2:5` is the sequence 2, 3, 4, and 5. Such an expression is often used in the `for` loops described in Section B.7.

B.5.2 Lists

Whereas the elements of vectors must all be of the same type, the elements of lists can be different types of objects. A simple example of this is

```
simple_list <- list("XXX", c(3.2, 7.1, 9), 42i)
```

where the elements are, of course, the string `"XXX"`, the vector `c(3.2, 7.1, 9)`, and the imaginary number `42i`. The elements of lists can be named as well, as in the following example:

```
my_list <- list(x = "XXX", y = c(3.2, 7.1, 9), z = 42i)
```

The components of this list can be accessed by numeric index or name. The first element `my_list[[1]]` (i.e., “`"XXX"`”) can also be accessed as `my_list[["x"]]`. Here, double brackets rather than single brackets are used instead. This is because single brackets can result in unexpected results when used with lists. For example, if one were to execute the following statement,

```
my_list[1] <- c(3,4,5)
```

this would produce the warning message “`number of items to replace is not a multiple of replacement length`” and replace the first element of `my_list` with 3, *not* the vector `c(3,4,5)`. In contrast,

```
my_list[[1]] <- c(3,4,5)
```

would do the intended operation of assigning the first element of `my_list` to the intended replacement value of `c(3,4,5)`.

One can use assignment to add elements to a list. For example,

```
new_list <- list()

new_list[[1]] <- 5
new_list[[2]] <- "trout"
```

creates a list with the elements `5` and `"trout"`.

Logical indexing can also be applied to lists. For example, `my_list[c(FALSE, TRUE, TRUE)]`, where `my_list` is as previously defined, returns a list containing the second and third elements of `my_list`. Double brackets do *not* work with logical indices, however; single brackets must be used.

The function `c` also works on lists as well as vectors. As mentioned before, this function returns a value that is a concatenation of all its arguments, with the type of value it returns depending on the type of its arguments. If the arguments are lists, then the return value is a list. For example, given the following R statements,

```
list1 <- list(x = 42, y = c(1,3,2))
list2 <- list(abc = 123)
```

the expression `c(list1, list2)` is a list equivalent to the expression `list(x = 42, y = c(1,3,2), abc = 123)`.

There are a few functions that are used to get properties of lists. The function `names` is used to retrieve the labels of a list. For example, `names(my_list)`, where `my_list` is as previously defined, returns `c("x", "y", "z")`. The function `length` returns the number of elements in a list. Other functions are used to invoke other functions on the elements of lists. For example, `lapply(my_list, length)` applies the function `length` to the elements of `my_list`, returning the list `list(x = 1, y = 3, z = 1)`, where the labels are the same as those of `my_list`, and the values associated with those labels are the lengths of the corresponding components of `my_list`. The function `sapply` operates similarly, but `sapply(my_list, length)` returns a vector rather than a list. The elements of this vector, of course, are still the lengths of the corresponding components of `my_list`, and the labels of that vector are the same as those of `my_list`.

B.5.3 Arrays

An array in R may be considered a generalization of a vector in some sense. Whereas a vector contains a sequence of values, an array contains an *n*-dimensional grid of values. The values in an array must all have the same type. A 2-D array that represents a 2×3 grid of values may be constructed as follows:

```
A <- array(c(1,2,3,4,5,6), dim = c(2,3))
```

The second argument of the `array` function, `dim`, indicates the dimensions of this array, which is 2×3. The first argument indicates the values stored in the array, which may be visually represented as the following table of values:

$$\begin{array}{ccc} 1 & 3 & 5 \\ 2 & 4 & 6 \end{array}$$

Accessing elements of an array is similar to accessing elements of a vector. For example, `A[1,2]` is the element in the first row and second column in the previous arrangement of numbers (i.e., 3). The expression `A[2,2:3]` indicates the second and third elements of the second row (i.e., 4 and 6). Also, `A[2,]` is the whole second row (i.e., 2, 4, and 6), and `A[,3]` is the third column (i.e., 5 and 6). Like vectors, array elements may be accessed via string labels. For example, if one executes the following R statement,

```
dimnames(A) <- list(c("one", "two"), c("A", "B", "C"))
```

then the rows and columns of the above table of values are labeled as follows:

	"A"	"B"	"C"
"one"	1	3	5
"two"	2	4	6

Accordingly, `A["one", "B"]` is the same as `A[1,2]`, and `A[,"C"]` is the same as `A[,3]`.

Logical indexing applies to arrays as well. For example, `A[c(TRUE, FALSE), c(TRUE, TRUE, FALSE)]` returns the first and second elements of the first row (1 and 3). This is because the logical index vector for the rows, `c(TRUE, FALSE)`, marks the first row as `TRUE`, while the logical index vector for the columns, `c(TRUE, TRUE, FALSE)`, has the first and second columns marked as `TRUE`.

Again, usually logical indexing is used with relational and/or logical operators. For example, the expression `A[1, A[1,] > 2]` returns the elements from the first row that are greater than 2.

Arithmetic operators also work with arrays mostly as they do with vectors. 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[1,1,1]*A2[1,1,1]`, `A1[1,1,2]*A2[1,1,2]`, and so on. If `b` is an ordinary number, then `A1^b` is such that its elements are `A1[1,1,1]^b`, `A1[1,1,2]^b`, and so on. The logical operators "`&`" and "`|`" also operate elementwise on arrays. Functions that are built into R often operate elementwise on arrays as well, just as they do on vectors.

There are a few functions that are used to get properties of arrays. One of them, the `dimnames`, has been shown before. In the use shown previously, one adds labels to an array `A` by assigning to `dimnames(A)`. However, in the reverse case, where

```
dimnames_A <- dimnames(A)
```

the function `dimnames` is used to retrieve the labels of `A` and store them in the variable `dimnames_A`. The function `dim` obtains the size and shape of an array. For example `dim(A)`, where `A` is defined as shown previously, returns `c(2,3)`. For 2-D arrays (which in R are also called matrices), the functions `nrow` and `ncol` return the number of rows and columns, respectively, and the functions `rownames` and `colnames` return the labels of rows and columns, respectively, as vectors of character strings.

B.5.4 Data Frames

Data frames 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 may be created as follows:

```
df <- read.table("my_data.csv", sep = ",", header = TRUE)
```

Here, the argument "`sep = ","`" indicates that a comma should be taken as the separator between two elements of a row, and "`header = TRUE`" indicates that the first line of the file represents column headers. Equivalently, the data frame may be read in the following way:

```
df <- read.csv("my_data.csv")
```

The elements in a data frame can be accessed using the same indexing methods used for 2-D arrays shown in Section B.5.3. Columns of a data frame can also be accessed much like the named elements of a list. For example, `df[["AA"]]` is a vector with the components 1.8, 3.1, 0.5, and 0.4.

There are a few functions that are used to get properties of data frames, and many of these are effectively the same as those used on 2-D arrays, such as `dimnames`, `dim`, `nrow`, `ncol`, `rownames`, and `colnames`. For data frames—but *not* arrays—the function `names` does the same thing as `colnames`.

B.6 Plotting Basics

The capabilities and limitations of R's plotting functionality may be shown with some simple examples. Suppose, for instance, that one wished to plot the following variables:

```
x <- 1:10
x_sq <- x^2
```

This may be done simply with the following R code:

```
pdf(file = file.path("plot_files", "plot_example1.pdf"),
    title = "Example plot 1", width = 3.0, height = 3.0,
    pointsize = 10)

plot(x, x_sq, xlab = "x", ylab = expression(x^2),
     type = "l", lty = 1, col = "blue")

dev.off()
```

The function `pdf` opens the ".pdf" file to which the plot is to be written. The `file` argument specifies the name of the file. (In the previous example, the directory `plot_files` is assumed to already exist.) The `title` argument specifies the title string that is embedded in the resulting ".pdf" file. While this string is mostly arbitrary, it is often shown in the titlebar of the window showing the ".pdf" file. The `width` and `height` arguments set the width and height of the plot to 3 inches, and the `pointsize` argument sets the default font size of the text in the plot to 10 points. In the call to the `plot` function, the `xlab` and `ylab` arguments indicate the labels for the *x*- and *y*-axes, and `ylab` is set to an R expression rather than a string in order to allow the y-axis label to have a superscript. The argument "`type = "l"`" indicates that a line rather than points are plotted; the argument "`lty = 1`" indicates that the line is solid (as opposed to dashed or dotted), and the argument "`col = "blue"`" indicates that the line is colored blue. Finally `dev.off()` closes the ".pdf" file. The resulting plot is shown in Fig. B-1a.

Suppose one wishes to plot the following variable as well:

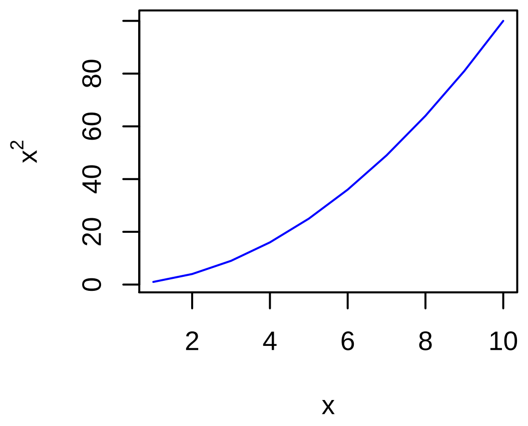
```
two_x_sq <- 2*x_sq
```

One might first attempt to plot `two_x_sq` and the previous variables `x` and `x_sq` on the same graph as follows:

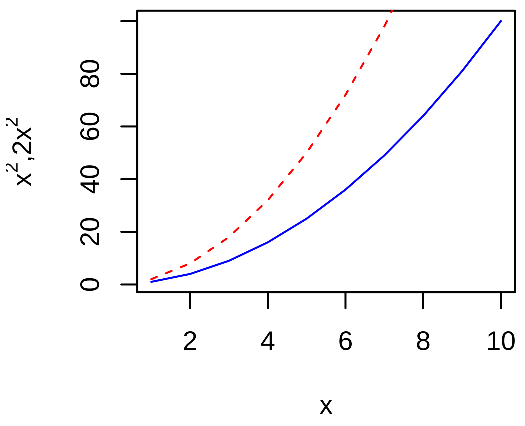
```
pdf(file = file.path("plot_files", "plot_example2.pdf"),
    title = "Example plot 2", width = 3.0, height = 3.0,
    pointsize = 10)

plot(x, x_sq, xlab = "x", ylab = expression(paste(x^2, ",", 2*x^2)),
     type = "l", lty = 1, col = "blue")

lines(x, two_x_sq, lty = 2, col = "red")

dev.off()
```

Here, the function `lines` is used to add an additional curve to the plot. The argument "`lty = 2`" indicates that this curve is plotted as a dashed line, and "`col = "red"`" indicates that the color of this curve is red. In the `plot` function, the `ylab` argument has become more complicated, with the `paste` function used to present the expressions `x^2` and `2*x^2` side-by-side, separated by a comma.

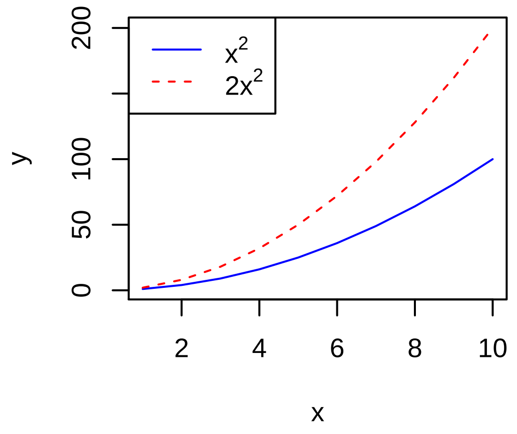
The resulting plot, shown in Fig. B-1b, has a couple problems. First, it is not clear which line belongs to which variable. Second, not all of the second line is plotted. The limits of the *y*-axis are based only on the range of `x_sq`, and are *not* readjusted when new curves are added to the plot. To fix the first problem, a legend is added in the following example. To fix the second problem, the limits of the *y*-axis are explicitly supplied:



(a)



(b)



(c)

Fig. B-1 Example plots used to illustrate the plotting features of R

```
pdf(file = file.path("plot_files", "plot_example3.pdf"),
    title = "Example plot 3", width = 3.0, height = 3.0,
    pointsize = 10)

line_types <- 1:2
line_colors <- c("blue", "red")

plot(x, x_sq,
     xlab = "x", ylab = "y", type = "l",
     lty = line_types[1], col = line_colors[1],
     ylim = range(c(x_sq, two_x_sq)))

lines(x, two_x_sq, lty = line_types[2], col = line_colors[2])

legend("topleft",
       legend = c(expression(x^2), expression(2*x^2)),
       lty = line_types, col = line_colors)

dev.off()
```

There are a couple things to note about the previous code, which results in the plot shown in Fig. B-1c. First, the limits for the *y*-axis are set with the `ylim` argument of the `plot` function. These limits are found via the function `range`, which returns a vector containing the minimum and maximum elements of its argument. Here, this argument is simply the concatenation of `x_sq` and `two_x_sq`. Second, the `legend` function *has* to have the line types and colors explicitly supplied to it, unlike the corresponding functions in MATLAB[3] and Matplotlib.[4]

B.7 Control Structures

Control structures allow for more complicated logic to be used in R scripts. A few of these structures are shown in this section.

B.7.1 Branching: `if` and `else`

The `if/else` structure is 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
}
```

[3]MathWorks, Inc. c2018 [accessed 2018 May]. `https://www.mathworks.com/products/matlab.html`.

[4]Matplotlib development team. c2018 [accessed 2018 May]. `https://matplotlib.org/`.

Here, the comments substitute for statements that would be used in an `if`/`else` structure in practice. 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 "`else if (...) {...}`" and "`else {...}`" clauses are optional. However, if either clause is used, then the closing brace "`}`" *must* precede the keyword `else`, or else R produces the error message "`unexpected 'else' in "else"`".

B.7.2 Iteration: `while` and `for`

The `while` loop is as follows:

```
while (condition) {
   # Statements executed so long as condition is TRUE
}
```

Here, the comment substitutes 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`. The statements in the brackets after `while(...)` 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 R 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 <- 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[2]

   i <- 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 above `while` loop can be rewritten with a `for` loop:

```
threshold <- 1e-6

for (i in 1:1000) {
   estimate_and_err <- do_estimate(A,B,C)
   err <- estimate_and_err[2]

   if (err <= threshold) {
      break
   }
}
```

A `for` loop is largely intended to execute a fixed number of iterations. Provided that `err` always exceeds `threshold`, this 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, 1 for the first iteration, 2 for the second, and so on, until `i` reaches the value of 1000. 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 the following,

```
for (i in 1:length(vec1)) {
   vec2[i] <- do_something(vec1[i])
}
```

the loop variable `i` is used to access successive values of the vectors `vec1` and `vec2`.

A `for` loop does not necessarily have to involve an expression such as `1:1000` or `1:length(vec1)`. Indeed, as indicated in Section B.5.1, `1:1000` is simply a vector consisting of the integers from 1 to 1000. Rather, one may iterate over any vector. For example, the following R code,

```
list_abD <- list(a = 2, b = 5, D = 7)
for (curr_name in names(list_abD)) {
  print(curr_name)
}
```

simply prints out the labels of the elements of `list_abD`, that is, `"a"`, `"b"`, and `"D"`.

B.8 External Sources and Packages

Suppose that one has written an R script with a set of functions that one would like to use in another script. The simplest solution, aside from just copying and pasting those functions into the next script, is to cut and paste those functions into a new file called, for example, `my-functions.R`, and then in both the old and new scripts, use the following statement,

```
source("my-functions.R")
```

in place of where those functions would be defined. This allows both scripts to use those functions, provided that they are in the same directory as `my-functions.R`.

Other developers and researchers have created suites of R functions to be used by other people, but rather than put them in an R file to be read in by the `source` function, these suites are organized into R *packages*. Several of these packages are available through Anaconda[5] or through the package manager of a Linux distribution. Most of these packages are also available via the Comprehensive R Archive Network (CRAN).[6] To install a package from CRAN, one may execute the statement

```
install.packages("name_of_package")
```

where `"name_of_package"` is a string with the name of the package. Once the package is loaded, it may be loaded with the following statement,

```
library(name_of_package)
```

In practice, R packages often not only contain R code but also code in compiled languages, such as C, C++, or Fortran, that is used internally by the R functions in the package. This code is compiled when the `install.packages` is used.

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

[6]R Foundation. Comprehensive R Archive Network. c2018 [accessed 2018 May]. `https://cran.r-project.org/`

B.9 Saving R Objects to Files

An object in R, such as one of the data structures described in Section B.5, can often be saved to a special file, called an R Data Serialization (RDS) file, that can be used to read in the object into another R session. The following code shows an example of saving an object:

```
xyz <- list(x = c(3,4,2), y = 3 + 2i, z = "string_val")

saveRDS(xyz, "xyz.rds")
```

The function `saveRDS` saves the list `xyz` to a file named `xyz.rds`. In another R session, the list can be read in with the `readRDS` function and printed as follows:

```
xyz_take_2 <- readRDS("xyz.rds")

print(xyz_take_2[["x"]])
print(xyz_take_2[["y"]])
print(xyz_take_2[["z"]])
```

As can be seen from the previous 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.

If the type of an object is part of the base R language, then the object can generally be saved to an RDS file. However, this is not necessarily true of objects whose types are defined in external packages. For example, if one attempts to save an object of the type `big.matrix` from the `bigmemory` package, the resulting RDS file just contains a pointer to the memory used by the `big.matrix` object, rather than the contents of that memory. The pointer becomes invalid once the R session ends, so the RDS file is useless.[7] Objects whose types are defined in other packages may have different limitations on whether they can be successfully stored in RDS files.

Also, in general, RDS files are meant for short-term storage, since there is no guarantee that the internal format of the RDS file will stay the same from one version of R to the next.

[7]How can I save and load a `bigmemory::big.matrix` object in R? c2015 [accessed 2018 May]. `https://stackoverflow.com/questions/32873859/how-can-i-save-and-load-a-bigmemorybig-matrix-object-in-r`

Appendix C. R Code for Bayesian Analysis

The following is the contents of `bayes-stress-strain-utils.R`, a source file containing R functions that have been written for Bayesian analyses of strength models. Comments in the file 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 tidyverse style guide.[1]

```r
#' Ensures that path to a file exists, and if not creates it.
#'
#' @param file_name String containing name of file
#' @return file_name, invisibly
ensure_path_to_file_exists <- function(file_name) {
  file_dir <- dirname(file_name)

  # Creating the directory that will contain the file, if that
  # directory does not yet exist.
  if (!dir.exists(file_dir)) {
    # If "recursive = TRUE" were not used here, dir.create would only
    # attempt to create the last directory component of file_dir.
    dir.create(file_dir, recursive = TRUE)
  }

  invisible(file_name)
}

#' 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.
#'
#' @param sigma_model_func 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 R list with named
#'   components)
#' @param epsilon_p_max Largest plastic strain for which stresses will
#'   be calculated
#' @param epsilon_p_dot Plastic strain rate
#' @param T_init Initial temperature of the sample being deformed
#' @param theta_model Model parameters of the strength model
#' @param beta_TQ Taylor-Quinney coefficient
#' @param rho Density of sample being deformed
#' @param specific_heat_func Function that returns the specific heat
#'   for a given temperature
#' @param curve_size Number of data points in the
#'   stress-strain curve
#' @return A list with the following named components:
#'
```

[1] Wickham H. The tidyverse style guide: code documentation. c2018 [accessed 2018 May]. http://style.tidyverse.org/code-documentation.html

```
#' * `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
#'
simulate_data <- function(sigma_model_func,
                          epsilon_p_max,
                          epsilon_p_dot,
                          T_init,
                          theta_model,
                          beta_TQ,
                          rho,
                          specific_heat_func,
                          curve_size) {
  #!{sndepstart}
  epsilon_p <- seq(0.0, epsilon_p_max, length.out = curve_size)
  #!{sndepend}

  #!{sndinitzerostart}
  temperature <- numeric(curve_size)
  sigma <- numeric(curve_size)
  #!{sndinitzeroend}

  #!{sndsetfirstelemstart}
  temperature[1] <- T_init

  sigma[1] <- sigma_model_func(
    epsilon_p[1],
    epsilon_p_dot,
    temperature[1],
    theta_model
  )
  #!{sndsetfirstelemend}

  #!{sndsetotherelemsstart}
  for (i in 2:curve_size) {

    # Estimate of area under stress-strain curve from
    # epsilon_p[i-1] to epsilon_p[i].
    area_under_curve <- sigma[i-1]*(epsilon_p[i] - epsilon_p[i-1])

    temp_rise <- beta_TQ*area_under_curve/
      (rho*specific_heat_func(temperature[i-1]))

    temperature[i] <- temperature[i-1] + temp_rise

    sigma[i] <- sigma_model_func(
      epsilon_p[i],
      epsilon_p_dot,
      temperature[i],
      theta_model
    )
  }
  #!{sndsetotherelemsend}

  #!{sndreturnstart}
  return (list(
      T = temperature,
      epsilon_p = epsilon_p,
      sigma = sigma
  ))
  #!{sndreturnend}
}

#' Make an empty plot with x and y axes
#'
#' This generates an empty plot window that can be drawn upon by
#' commands such as `lines()`, `points()`, etc. This is useful if
#' there are multiple superimposed plots.
#'
#' @param xlim Two-element numeric vector with the minimum and maximum
#'   of x values in plot
#' @param ylim Two-element numeric vector with the minimum and maximum
#'   of y values in plot
#'
make_empty_xy_plot <- function(xlim, ylim) {
  plot.new()  # This creates an empty plot "window". If the plot is
              # not being written to a file, this creates an actual
              # window in a graphical user interface.

  # This sets the ranges of x- and y-values shown in the plot.
  plot.window(xlim = xlim, ylim = ylim)

  axis(1) # Adding x-axis
  axis(2) # Adding y-axis

  box() # Adding a box that contains the actual contents of the plot
}

#' Plots stress-strain curves
#'
#' @param output_pdf String containing the name of the ".pdf" file
#'   containing the plots
#'
#' @param epsilon_p_dot Vector where element `i` contains the strain
#'   rate for curve `i`
#'
#' @param T_init Vector of where element `i` contains the initial
#'   sample temperature for curve `i`, must have same length as
#'   `epsilon_p_dot`
#'
#' @param epsilon_p List of vectors of strain values for all curves,
#'   where `epsilon_p[[1]]` contains the strain values for the first
#'   curve, `epsilon_p[[2]]` contains the strain values for the second
```

```r
#'   curve, etc.
#'
#' @param sigma List of vectors of stress values for all curves, where
#'   `sigma[[1]]` contains the stress values for the first curve,
#'   `sigma[[2]]` contains the stress values for the second curve,
#'   etc.
#'
#' @param space_for_legend Number between zero and one indicating the
#'   amount of space in the plot for the legend, such that, for
#'   example, space_for_legend = 0.2 increases the vertical size of
#'   the plot by 20%
#'
#' @param point_period integer value indicating what data points will
#'   be plotted. If the data points are so densely spaced as to
#'   overlap with each other, then one may wish to set `point_period`
#'   to a value larger than 1 in order to reduce the number of points
#'   shown.
#'
#' @param sigma_unit String specifying the units of stress, e.g.,
#'   "MPa"
#'
#' @param epsilon_p_dot_time_unit String (to be converted to R
#'   expression) specifying the units of time used in the strain rate,
#'   e.g., "s" or "sec" for a strain per second.
#'
#' @param T_init_unit String (to be converted to R expression)
#'   specifying the units of the initial sample temperature, e.g., "K"
#'   for Kelvin
#'
#' @param epsilon_p_label String (to be converted to R expression)
#'   specifying symbol used for strain
#'
#' @param sigma_label String (to be converted to R expression)
#'   specifying symbol used for stress
#'
#' @param epsilon_p_dot_label String (to be converted to R expression)
#'   specifying symbol used for strain rate
#'
#' @param T_init_label String (to be converted to R expression)
#'   specifying symbol used for temperature
#'
#' @return The string `output_pdf`, invisibly
plot_stress_strain_curves <- function(output_pdf,
                                      epsilon_p_dot,
                                      T_init,
                                      epsilon_p, sigma,
                                      space_for_legend = 0.2,
                                      point_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]") {

  # The number of stress-strain curves, num_curves, should be the
  # same as the number of elements in epsilon_p_dot.
  num_curves <- length(epsilon_p_dot)

  # Setting up line types, point types, color --------------------
  # values and legend labels to be used in the -------------------
  # actual plot. -------------------------------------------------

  # R has six possible line types, numbered 1 to 6. (There is also a
  # "invisible" line type, i.e. no line at all, numbered as 0, but it's
  # not used here.)
  poss_line_types <- 1:6

  # These are the line types used. Note that if num_curves is greater
  # than max(poss_line_types), some line types will be recycled.
  line_types <- rep(poss_line_types, length.out = num_curves)

  # R has several numbered point types. Here, point types 0 through 20
  # are used. NA is the invisible point type, i.e., no point plotted.
  poss_pt_types <- c(NA, 0:20)
  #
  # (Note: numbered point types 21 through 25 aren't used here because
  # they are nearly identical to some of the previous point types, but
  # require an additional plotting parameter to set their color.)

  # This tiles the possible point types in a particular
  # fashion. First, the first point type (i.e., NA) is repeated up to
  # the number of possible line types, then the second point type
  # (i.e., 0) is repeated up to the number of possible line types,
  # etc., until a vector of length num_curves is created.
  pt_types <- rep(
    poss_pt_types,
    each = length(poss_line_types),
    length.out = num_curves
  )

  # This creates a vector of length num_curves, containing color
  # values from the current palette.
  color_vals <- rep(palette(), length.out = num_curves)

  # This initializes the vector that will contain the text and math
  # expressions in the legend.  Each element of this vector contains
  # the label for each curve in the legend.
  legend_labels <- rep(NA, num_curves)

  # Creating the actual plots ------------------------------------

  ensure_path_to_file_exists(output_pdf)

  # This causes the plots to be written to a ".pdf" file.
  pdf(
    file = output_pdf, # This indicates that the name of the ".pdf"
```

```
                      # file will be "output_pdf".
  title = basename(output_pdf) # This indicates the title that
                               # will be embedded in the resulting
                               # ".pdf" file. While this is mostly
                               # arbitrary, the title will often
                               # be shown in the titlebar of the
                               # window showing the ".pdf" file.
)

# The variables xrange and yrange will be used to set the ranges of
# x- and y-values shown in the plot. The function "unlist" takes a
# list of vectors and returns one long vector containing all the
# elements of the vectors in the list. The function "range" returns
# the minimum and maximum elements of its vector argument.
xrange <- range(unlist(epsilon_p))
yrange <- range(unlist(sigma))

# Making room for the legend at the bottom of the plot
yrange[1] <- yrange[1] - space_for_legend*(yrange[2] - yrange[1])

# Initializing the plot window --------------------------------
make_empty_xy_plot(xrange, yrange)

# Adding line plots to the plot window -------------------------

for (curve_ind in 1:num_curves) {

  # This adds a stress-strain curve to the plot with the line type
  # indicated by "lty" and the color indicated by "col".
  lines(
    epsilon_p[[curve_ind]],
    sigma[[curve_ind]],
    lty = line_types[curve_ind],
    col = color_vals[curve_ind]
  )

  # This causes a point to be printed every "point_period" along the
  # stress-strain curve, with point style pt_types[curve_ind] and
  # color color_vals[curve_ind]. If there are so many data points
  # that the points would overlap if point_period = 1, then
  # point_period can be set to a higher value to reduce the number
  # of points printed.
  curve_size <- length(epsilon_p[[curve_ind]])

  points(
    epsilon_p[[curve_ind]][seq(1, curve_size, point_period)],
    sigma[[curve_ind]][seq(1, curve_size, point_period)],
    pch = pt_types[curve_ind],
    col = color_vals[curve_ind]
  )

  # This creates a label to be used in the legend for this curve.
  legend_labels[curve_ind] <- parse( # For what "parse" means, see
                                     # the comment below.
    text = sprintf(
      "paste(%s, ' = %g ', %s, ', ', %s, ' = %g/', %s)",
      T_init_label,
      T_init[curve_ind], T_init_unit,
      epsilon_p_dot_label,
      epsilon_p_dot[curve_ind], epsilon_p_dot_time_unit
    )
  )
  # The function "parse" used above actually creates an R
  # expression. The content of this R expression follows a syntax
  # that is found in the section "Mathematical Annotation in R" of
  # the R reference documentation. Type "help(plotmath)" without the
  # quotes to see this documentation.
}

# Adding the legend and title -----------------------------------

# This addes the legend to the plot. The first argument to this
# function is the position of the legend in the plot. Next to the
# label legend_labels[i] in the legend will be a segment of the line
# with line type line_types[i], with a point of type pt_types[i],
# and color color_vals[i].
legend(
  "bottomright",
  legend = legend_labels,
  lty = line_types,
  pch = pt_types,
  col = color_vals,
  ncol = ifelse(num_curves > 4, 2, 1) # This indicates that the
                                      # legend will be shown with
                                      # two columns if there are
                                      # more than four stress-strain
                                      # curves.
)

# This adds labels to the x- and y- axes.
title(
  xlab = parse(text = epsilon_p_label),
  ylab = parse( # For what "parse" means, see the comment below.
    text = sprintf("paste(%s, ' (%s)')", sigma_label, sigma_unit)
  )
)
# The function "parse" used above actually creates an R
# expression. The content of this R expression follows a syntax that
# is found in the section "Mathematical Annotation in R" of the R
# reference documentation. Type "help(plotmath)" without the quotes
# to see this documentation.

# Final "clean up" ----------------------------------------------

dev.off() # Closing the plot "window"

# This returns the name of the ".pdf" file. Without this, this
# function would return the output of "dev.off()", which would be
```

```
  # mostly harmless but look wrong, especially in a Jupyter notebook.
  invisible(output_pdf)
}

#' Generate a function that linearly interpolates tabular data
#'
#' @param tab_data_file File with tabular data
#' @param x_col Column of the tabular data that contains the x-values,
#'   defaults to 1
#' @param y_col Column of the tabular data that contains the y-values,
#'   defaults to 2
#' @param conv_func_x Function applied to all x-values in the tabular
#'   data, defaults to the identity function
#' @param conv_func_y Function applied to all y-values in the tabular
#'   data, defaults to the identity function
#' @param ... Keyword arguments for the `read.table` function, which
#'   is used to read the tabular data
#' @return A function that returns a linear interpolation of y-values
#'   given an x-value
#'
gen_lin_interp_func <- function(tab_data_file,
                                x_col = 1,
                                y_col = 2,
                                conv_func_x = function(x) {x},
                                conv_func_y = function(y) {y},
                                ...) {

  xydata <- read.table(tab_data_file, ...)

  return (approxfun(conv_func_x(xydata[,x_col]),
                    conv_func_y(xydata[,y_col]), rule = 2))
}

#' Wrapper for saveRDS
#'
#' This wrapper ensures that if there are any directory components in
#' the path to the RDS file to be written, these components will
#' exist. If they do not yet exist, they will be created.
#'
#' @param obj Object to be written to an RDS file
#' @param rds_file_name Path of the RDS file to be written
#' @return `NULL`, invisibly
#'
save_to_rds <- function(obj, rds_file_name) {
  ensure_path_to_file_exists(rds_file_name)
  saveRDS(obj, rds_file_name)
}

#' Save summary statistics and MCMC samples to CSV files
#'
#' @param fit `stanfit` object
#' @param summary_csv_filename Name of CSV file to which summary
#'   statistics will be written
#' @param samples_csv_filename Name of CSV file to which MCMC samples
```

```r
#'   will be written. If the file ends in ".gz", it will be
#'   Gzip-compressed.
#'
save_stan_fit_to_csv <- function(fit,
                                 summary_csv_filename,
                                 samples_csv_filename) {

  #!{ssftcpathstart}
  ensure_path_to_file_exists(summary_csv_filename)
  ensure_path_to_file_exists(samples_csv_filename)
  #!{ssftcpathend}

  #!{ssftcpathsumwritestart}
  write.csv(summary(fit)[["summary"]], summary_csv_filename)
  #!{ssftcpathsumwriteend}

  #!{ssftcpathgzipstart}
  if (endsWith(samples_csv_filename, ".gz")) {
    out_file <- gzfile(samples_csv_filename, "w")
  } else {
    out_file <- file(samples_csv_filename, "w")
  }

  # Makes sure that out_file is closed, even if something goes wrong
  # in write.csv.
  on.exit(close(out_file))
  #!{ssftcpathgzipend}

  #!{ssftcpathsampwritestart}
  write.csv(as.matrix(fit), out_file, row.names = FALSE)
  #!{ssftcpathsampwriteend}
}

#' Calculate the temperatures for the data points along a stress-strain curve
#'
#' @param T_init Initial temperature
#' @param epsilon_p Sequence of plastic strains
#' @param sigma Sequence of stresses the same length as epsilon_p
#' @param f_area A fraction such that f_area*sigma[1]*epsilon_p[1] is
#'   a reasonable estimate of the area under the missing part of the
#'   stress-strain curve over the interval [0, epsilon_p[1]].
#'   Generally, f_area should be greater than 0.5, but if epsilon_p[1]
#'   is zero, then f_area should be set to zero.
#' @param beta_TQ Taylor-Quinney coefficient
#' @param rho Density of sample being deformed to obtain stress-strain
#'   curve
#' @param specific_heat_func A function accepts a temperature and
#'   returns a specific heat
#' @return Sequence of temperatures such that T[i] is the temperature
#'   for data point (epsilon_p[i], sigma[i])
#'
calc_temps <- function(T_init, epsilon_p, sigma,
                       f_area, beta_TQ, rho, specific_heat_func) {
```

```r
  #!{ctinitstart}
  curve_size <- length(epsilon_p)
  temperature <- numeric(curve_size)

  temperature[1] <- T_init + beta_TQ*f_area*sigma[1]*epsilon_p[1]/
    (rho*specific_heat_func(T_init))
  #!{ctinitend}

  #!{ctcalcstart}
  for (i in 2: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(temperature[i-1]))

    temperature[i] = temperature[i-1] + T_rise
  }

  return (temperature)
  #!{ctcalcend}
}

#' Plot histogram (on pre-existing plot window) with segments that
#' outline the shapes of the bars that would normally be shown in a
#' histogram
#'
#' @param hist_obj object of class "histogram" returned by `hist()`
#' @param freq Same meaning as the `freq` argument of `hist()`
#' @param ... Keyword arguments passed to `segments` function used to
#'   draw steps, such as `col`, `lty`, etc.
#'
#' @examples
#'
#' x_hist <- hist(x, plot = FALSE)
#' make_empty_xy_plot(range(x_hist[["breaks"]]), range(x_hist[["density"]]))
#' hist_outline(x_hist)
hist_outline <- function(hist_obj, freq = FALSE, ...) {
  if (freq) {
    hist_vals <- hist_obj[["counts"]]
  } else {
    hist_vals <- hist_obj[["density"]]
  }

  num_bins <- length(hist_vals)

  # Horizontal histogram segments
  segments(
    hist_obj[["breaks"]][1:num_bins],
    hist_vals,
    hist_obj[["breaks"]][2:(num_bins + 1)],
    hist_vals,
    ...
  )

  # Vertical histogram segments
  segments(
    hist_obj[["breaks"]],
    c(0.0, hist_vals),
    hist_obj[["breaks"]],
    c(hist_vals, 0.0),
    ...
  )
}

#' Plot a region of color betwen two curves
#'
#' @param x x-coordinates of each curve
#' @param y1 y-coordinates of the first curve
#' @param y2 y-coordinates of the second curve
#' @param col Color of filled region
#'
#' @examples
#'
#' x <- 1:9
#' y1 <- x^2
#' y2 <- 2*y1
#' make_empty_xy_plot(range(x), range(c(y1, y2)))
#' fill_between_curves(x, y1, y2)
fill_between_curves <- function(x, y1, y2, col = "gray") {
  x_poly <- c(x, rev(x))
  y_poly <- c(y1, rev(y2))

  polygon(x_poly, y_poly, border = NA, col = col)
}
```

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;
    }
  }
}
```

List of Symbols, Abbreviations, and Acronyms

β_{TQ}	Taylor-Quinney coefficient
$\dot{\epsilon}_p$	plastic strain rate
$\dot{\epsilon}_{p0}$	reference plastic strain rate, $1/\mathrm{s}$
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
2-D	two-dimensional
3-D	three-dimensional

ARL	CCDC Army Research Laboratory
BCC	body-centered cubic
CRAN	Comprehensive R Archive Network
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
NUTS	no-U-turn sampler
PFP	pushed forward posterior
PPD	posterior predictive distribution
RDS	R Data Serialization
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