SUPPLEMENTARY MATERIAL FOR : Thermal weakening friction and seismic slip: an efficient numerical scheme for thermal diffusion

S. Nielsen[1], E. Spagnuolo[2], M. Violay[3]and G. Di Toro[4]

[1]Department of Earth Sciences, Durham University, *DH1 3LE* Durham, United Kingdom
[2]Istituto nazionale di Geofisica e Vulcanologia, Via di Vigna Murata, 605, *00143* Roma, Italy
[3]EPFL ENAC IIC LEMR, GC D1 401 (BÃ¢timent GC), Station 18, *CH-1015* Lausanne, Switzerland
[4]Dipartimento di Geoscienze, via G. Gradenigo, 6, *35131* Padova, Italy

Additional tests of friction versus experimental data

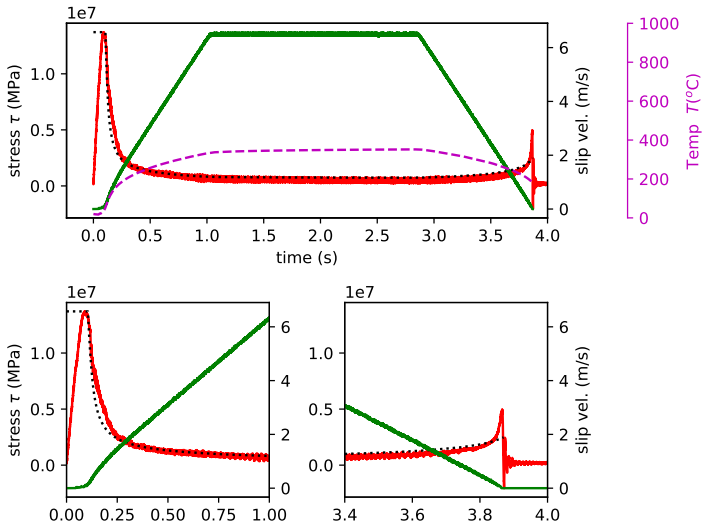


Fig. S1. Experimental fit of friction in Carrara marble with Arrhenius thermal weakening and heat sinks (experiment s308, normal stress 20 MPa). Solid red: measured experimental shear stress. Dotted black: model friction. Solid green: measured slip velocity. Dashed purple: computed temperature.

Corresponding author: Stefan Nielsen, stefan.nielsen@durham.ac.uk

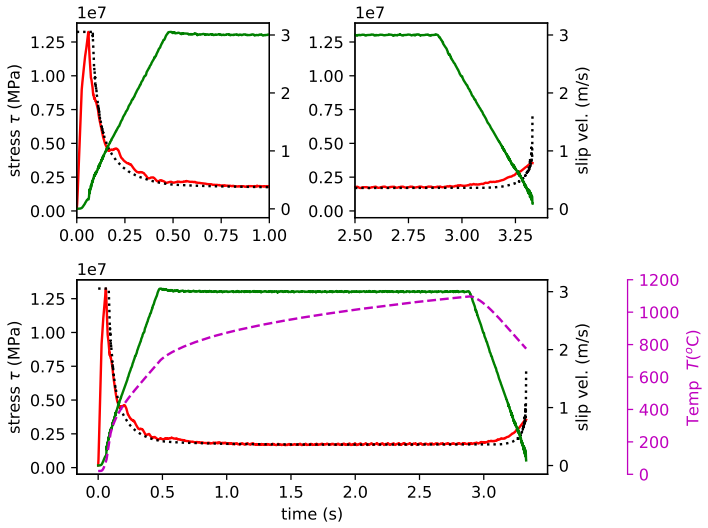


Fig. S2. Experimental fit of friction in gabbro with FWSS (normal stress 20 MPa). Solid red: measured experimental shear stress. Dotted black: model friction. Solid green: measured slip velocity. Dashed purple: computed temperature. In this example it can be seen that the recovery is not well reproduced.

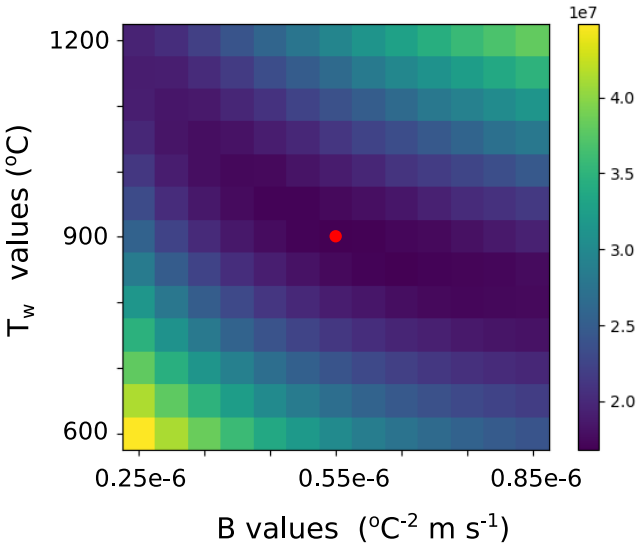


Fig. S3. Misfit computed between model and experimental frictional curve, for a grid of T_w, B parameters. The minimum misfit or best model in the sense of least squares is indicated by the red dot. Note the diagonal alignment of relatively low misfit values, indicating a trade-off between T_w and B. Most reasonable models lie along that diagonal spine for the intervals $800 < T_w < 1000$ and $0.45 < B < 0.65$. For this model the other relevant parameters were fixed as $\tau_w = 0.5$E6 Pa, $T_s = 873\ ^o$C, $C_s = 2$E6 W m^{-2}, $\kappa = 0.821$E-6 m^2 s, $\rho = 2700$ kg m^{-3}, $c = 833$ J K^{-1}. Misfit is defined here as $\sum_{n=1}^{N}(\tau_{comp}(n) - \tau_{exp}(n))^2$ where N is the number of time steps in the experimental and the numerical curve.

Thermal dependence of κ, ρ

We fit experimental data reported in Merriman (2018)[1] to derive an empirical law for thermal dependence of κ, c such that:

$$\begin{array}{rcll} \kappa(T) & = & A_d\ \exp(-T/B_d)+C_d & \text{(diffusivity)} \\ k(T) & = & A_c\ \exp(T/B_c)+C_c & \text{(conductivity)} \\ c(T) & = & \dfrac{k(T)}{\rho\ \kappa(T)} & \text{(capacity)} \end{array} \tag{1}$$

where $(A_d, B_d, C_d) = (0.534, 170.0, 0.288)$ and $(A_c, B_c, C_c) = (1.057, 292.3, 0.70)$ and $\rho = 2700$ kg m^{-3} is approximated as constant. Note that here the fit parameters $A_d, B_d, C_d, A_c, B_c, C_c$ assume T is the difference from room temperature of $T_r \approx 27^o$C (the experiments were conducted in Roma, Italy and it gets pretty warm in the summer).

The thermal diffusion is computed by finite differences (Crank-Nicholson method) with explicit spatial steps, and the parameters at each point of the grid are re-evaluated after each temperature update.

NUMERICAL CODES FOR TEMPERATURE (PYTHON):

Efficient wavenumber summation example:

```
# INITIALIZATIONS:
import numpy as np
import matplotlib.pyplot as plt
import time as tm
%matplotlib notebook
#
ttot=20.;
dt=ttot/1000;
nt=int(ttot/dt)
rho = 3000; c = 715; k = 1.1e-6;
M = 32
zmax =  (M/2.5) * np.sqrt(k * ttot);
zmin = (1/2) * zmax/M;
smax=2*np.pi/zmin
ds = smax/M;
smin=ds/2
s=np.asarray([(m - 1/2)*ds for m in range(1,M+1)],dtype=float)
theta=np.asarray([0.0 for M in range(0,M)],dtype=float)

# DEFINE THE TEMPERATURE COMPUTATION ROUTINES
# wavenumber summation:
def comp_temp_wav(q, theta):
    qn= q/(rho*c); Twav=0.
    for mm in range(0,M):
        theta[mm]= (dt * qn + theta[mm])/(1.0 + dt * k * s[mm]**2 ) # Back Euler
        Twav=Twav+theta[mm]*ds
        Twav=Twav*(2.0/np.pi)
    return(Twav,theta)
# classic time summation:
def comp_temp_sum(q):
```

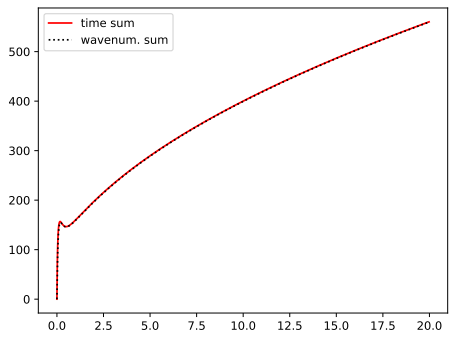
[1] J. D. Merriman; A. M. Hofmeister; D. J. Roy; . G. Whittington, Temperature-dependent thermal transport properties of carbonate minerals and rocks Geosphere (2018) 14 (4): 1961?1987. https://doi.org/10.1130/GES01581.1

```
    Tsum=0.
    for p in range(it):
    Tsum=Tsum + dt*q[p]/np.sqrt(dt * (float(it)-1/4) - dt * p)
    Tsum=Tsum * (1/(2 * rho * c * np.sqrt(k*np.pi)))
    return(Tsum)


# COMPUTE TIME EVOLUTION:
T=[0];Tb=[0];time=[0];theta[:]=0.0;
# imposed heat flow:
q=[5e5+3e6*np.exp(-dt*float(ii)/.1) for ii in range(nt)]
#
# compute with wavenumber summation:
start1=tm.time()
for it in range(1,nt):
    time.append(float(it)*dt)
    Twav,theta=comp_temp_wav(q[it-1]/2., theta)
    T.append(Twav)
    end1=tm.time()
## compute with classic time summation (OPTIONAL):
#start2=tm.time()
#for it in range(1,nt):
#    Tsum=comp_temp_sum(q)
#    Tb.append(Tsum)
#end2=tm.time()
#print(end1-start1,end2-start2, (end1-start1)/(end2-start2))
```

Plotting both solutions together:

```
fig,ax=plt.subplots()
ax.plot(time, Tb, 'r-',label='time sum')
ax.plot(time, T,'k:', label='wavenum. sum');
ax.legend()
```



Finite difference scheme

This is used for cases where the diffusivity and heat capacity are temperature-dependent, which are updated in comp_varpar.

The temperature solution uses a Crank-Nicholson finite difference method, which is illustrated below in the python function cofindi_var_b. At each time iteration cofindi_var_b is called to update the temperature with a new imposed flow value.

```
##################################
# functions and sub-functions definitions:
##################################

import numpy as np

def cofindi_var_b(alambda,ro,stept,stepz,T0,g):
        Ad,Bd,Cd=(0.534, 170.0, 0.288)
        Ac,Bc,Cc=(1.057, 292.3, 0.70)

        def comp_varpar(TT,rho):
                kv=1e-6 * ( Ad * np.exp(-TT / Bd) + Cd) #diffusiity
                cv=(Ac * np.exp(-TT/Bc)+Cc) / ((1e-6*(Ad * np.exp(-TT / Bd) + Cd))*rho) #heat capa
                return(kv,cv)

        def update_coeffs(alfa,ro,cp,alambda,stept,stepz):
                # various coefficients to update for use in solution
                kappar=alfa*ro*cp
                arf=(-kappar*alambda)
                brf=2*(ro*cp+kappar*alambda)
                crf=(-kappar*alambda)
                darf=(kappar*alambda)
                dbrf=2*(ro*cp-kappar*alambda)
                dcrf=(kappar*stept/stepz**2)
                return(kappar,arf,brf,crf,darf,dbrf,dcrf)
###################################
        # AT LOWER LIMIT (flow condition)
        #*********************************************************************
        i=1
        ###
        alfa,cp=comp_varpar(w[i],ro)
        kappar,arf,brf,crf,darf,dbrf,dcrf=update_coeffs(alfa,ro,cp,alambda,stept,stepz)
        #
        ai[i]=0
        bi[i]=arf+brf
        ci[i]=crf
        di[i]=(arf-darf)*g*stepz/kappar + (dbrf+darf)*w[i] + dcrf*w[i+1]
        # INSIDE THE MEDIUM (diffusion)
        for i in range(2,nzeff-2):
                ###
                alfa,cp=comp_varpar(w[i],ro)
                kappar,arf,brf,crf,darf,dbrf,dcrf=update_coeffs(alfa,ro,cp,alambda,stept,stepz)
                ###
                ai[i]=arf
                bi[i]=brf
                ci[i]=crf
                di[i]=darf*w[i-1] +  dbrf*w[i]  + dcrf*w[i+1]
        # AT UPPER LIMIT:
        i=nzeff-2
        ###
        alfa,cp=comp_varpar(w[i],ro)
        kappar,arf,brf,crf,darf,dbrf,dcrf=update_coeffs(alfa,ro,cp,alambda,stept,stepz)
        ###
        ai[i]=arf
```

```
        bi[i]=brf-crf
        ci[i]=0
        di[i]=-2*T0*crf+2*T0*dcrf+(darf)*w[i-1]+(dbrf-dcrf)*w[i]
        imin=1;imax=nzeff-2
        betai[imin]=bi[imin]
        #print(betai[imin]); sys.exit()
        gammai[imin]=di[imin]/bi[imin]
        for k in range(imin+1,imax+1):
                betai[k]=bi[k]-ai[k]*ci[k-1]/betai[k-1]
                gammai[k]=(di[k]-ai[k]*gammai[k-1])/betai[k]
        tpz[imax]=gammai[imax]
        # TRIDIAGONAL Matrix sol (Thomas algorithm):
        for k in range(1,imax-imin+1):
                kk=imax-k
                tpz[kk]=gammai[kk]-ci[kk]*tpz[kk+1]/betai[kk]
        #  ASSIGN TEMPERATURES
        for i in range(1,nzeff-1): #assegnazione
                w[i]=tpz[i]
        return(w[1])
##########################################
# Initialise arrays and parameters:
##########################################
nzmax=50 # number of steps in z --whatever is sufficient to avoid reflections
stepz=2.0e-4;stept=1.0e-2
nzeff=int(1.0*nzmax)-1
alambda=stept/stepz**2
nt=int(rdur/stept)
ai=np.zeros(nzmax); bi=np.zeros(nzmax); ci=np.zeros(nzmax);
di=np.zeros(nzmax); z=np.zeros(nzmax) # various coefficients and parameters
betai=np.zeros(nzmax); gammai=np.zeros(nzmax);
w=np.zeros(nzmax); tpz=np.zeros(nzmax)     # w is temperature
rdur=8.0 #7.0    !duration of experiment
T0=0.0
ro= 2.7E3             # !ro: density of rock

# g is heat flow at boundary, kappar is conductivity, ro is mass density,
# cp is heat capacity, alfa is diffusivity, T0 is zero here,
# alambda is stept/stepz**2,
# stepz and stept are space and time sampling
# rule of thumb diffusivity*stept/stepz**2 < 1/2
# The returned value w[1] is the temperature at the boundary.
# Inner temperatures are w [i] with i != 1.
#
# compute temperature at each time step:
########################################
T=[]
for it in range(nt):
        q(it)= #.... replace with heat flow at time "it" ....
        T.append( cofindi_var_b(alambda,ro,stept,stepz,T0,-q) )
```

NUMERICAL CODE FOR TEMPERATURE (FORTRAN):

```
c program cotemp2
c computes temperature for an imposed heat flow in 2 different ways
c Stefan Nielsen July 2020
parameter (M=32,nt=1000)
real*16 k,rho,c,tmax,zmax,zmin,smax,smin,ds,dt,qn,Tb,Tbc,T,pi
real*16 theta(M),s(M)
real*16 q(nt)
integer it,mm
c initializations
k=1.1e-6
rho=3000.
c=715.0
pi=3.1416
dt=0.02
tmax=float(nt)*dt
zmax=float(M)*(2./5.)*sqrt(k*tmax)
zmin=zmax/(2.*float(M))
smax=2.*pi/zmin
ds=smax/float(M)
smin=ds/2.
do mm=1,M
    s(mm)=(float(mm)-0.5)*ds
    theta(mm)=0.0
enddo
c pre-compute values of imposed heat flow:
do it=1,nt
    q(it)=3.0e6*exp(-float(it)*dt/.1) + 0.5e6
enddo
c COMPUTING WITH WAVE NUMBER SUMMATION:
open (22,file='tew.csv',form='formatted')
do it=2,nt !start time loop
    qn=q(it-1)/(2*rho*c)
    Tb=0.
    do mm=1,M ! start memory variables loop
        theta(mm)= dt * qn + theta(mm) * (1. - dt * k * s(mm)**2)
        Tb=Tb+theta(mm)*ds
    enddo ! end memory variables loop
    T=(2./pi)*Tb
    write (22,*) float(it)*dt, T
enddo ! end time loop
close (22)
cc COMPUTING WITH CLASSIC TIME SUMMATION (OPTIONAL):
c open (23,file='tes.csv',form='formatted')
c write (23,*) 0.0, 0.0
c cfact=(1./(2. * rho * c * sqrt(k*pi)))
c do it=2,nt ! start time loop
c    Tbc=0.
c    do j=1,it-1 ! summation over past times
c        Tbc=Tbc+cfact*dt*q(j)/sqrt(dt*(float(it)-float(j)-0.25))
c    enddo
c    write (23,*) float(it)*dt, Tbc
c enddo ! end time loop
c close(23)
```

```
stop
end
```