Numerical Simulation of Two-phase Heat Flow and Transient Partial Melting in the Lower Crust with Adaptive Mesh Refinement

Soon Hoe Lim

Thesis Advisor: Eric Hetland

Department of Physics

University of Michigan, Ann Arbor

April 5, 2013

Abstract

We have developed a 2D finite element program based on the finite element software library `deal.II` in order to investigate transient partial melting of the lower crust following intrusion of a single sill. Transient melting of the crust is crucial to the onset of migration and segregation of melt out of the lower crust to the Earth surface, and thus affects the thermal evolution of crust. We numerically solve for temperature and melt fraction in the governing time dependent heat balance equation using an adaptive h-mesh refinement scheme in a structured finite element mesh. The numerical techniques employed for the modeling includes space-time discretization using continuous Galerkin method for space and finite difference method for time, an iterative approach to the nonlinear problem, and parameter experiments to determine optimal adaptive remeshing strategy. Our study has identified two primary parameters that control the degree of transient melting of the crust: the initial crustal temperature and the dimensions of the intruded sills. Both parameters are related to the productivity of partial crustal melt through a power law.

Keywords : finite element method, space-time discretization, adaptive mesh refinement, intrusion of sill, transient partial melting, migration and segregation of melt

An honors thesis submitted in partial fulfillment of the requirements for an Honors degree of B.S. in Physics at the University of Michigan, Ann Arbor

Contents

Acknowledgements

I would like to thank many people who have helped and guided me in writing my honors thesis. Firstly, I would like to express special gratitude to my thesis advisor, Prof. Eric Hetland, for giving me the opportunity to work with him for the past 12 months in his geophysics lab. I would like to thank him for his patient and selfless guidance of my research and thesis writing, as well as his encouragement and support.

I would also like to thank Prof. Wolfgang Bangerth, Prof. Timo Heister, and other `deal.II` developers for making `deal.II` library available for writing the finite element program. The materials in this thesis are far from original. I acknowledge that a large parts of the codes in the program and the numerical methods are inherited from the tutorial programs published in the documentation for `deal.II` and my primary contribution has been to borrow, assimilate, and modify the data structures and algorithms provided by the library in order to solve a problem in an application. I have relied heavily on the tutorial programs and the online compact course organized by Prof. B. Janssen and Prof. T. Wick from the Institute of Applied Mathematics at University of Heidelberg in learning the finite element method and writing the program.

I would also like to thank other `deal.II` users in `deal.II` users forum and other members in the geophysics lab for providing me constructive hints and feedbacks in the process of the program development. Lastly, I would like to thank the Department of Physics at the University of Michigan for allowing me to complete and publish my honors thesis.

List of Figures

List of Tables

1 Introduction

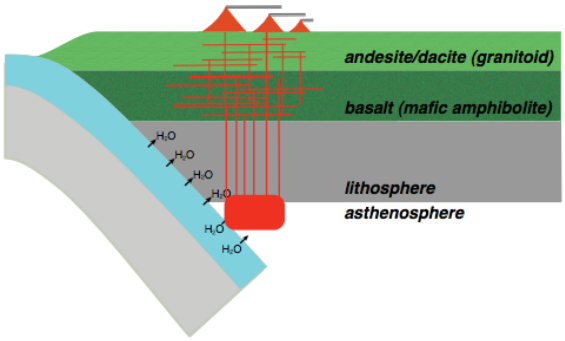


Figure 1: Conceptual illustration of the problem setting: A sill is emplaced at the lower crust, causing partial crustal melting and advection of melt (from [6]).

The temperature distribution in arc crust is driven by emplacement of sills into the lower crust, along with subsequent advection of melt up the crustal column or erupted to the surface [4]. The advected melt could be residual melt from the solidifying sills or partial melt of the crust surrounding the sills. Depending on the size of the sills and the background temperature in which the sills are intruding, a significant portion of the lower crust may be partially molten and sustain melt fractions higher than 20%. One of the consequences of such high melt fraction is migration and segregation of melt over large distances [9], resulting in crustal partial melt possibly either advected up the crustal column or erupted to the surface.

In this paper, we develop numerical models of temperature and melt fraction in order to investigate transient partial melting of the crust surrounding an intruded sill during the solidification of the sill. For simplicity, we only consider the case of a single sill emplacement in the lower crust at a specified background temperature. Specifically, the numerical models in this study investigate the effects of varying size of emplaced sills into crust with different background temperatures on the domain solidification time and evolution of crustal melt fraction.

The models are developed using `deal.II 7.1.0` library, or Differential Equations Analysis Library, a `C++` program library aimed at the computational solution of partial differential equations using adaptive finite elements [2]. We utilize space-time continuous Galerkin method and local h-adaptive refinement techniques in a structured mesh. The source codes of the developed program, which contain the files `source.cc`, `parameter.prm`, and `Makefile` are included in Appendix B for reference. From this investigation, we hope that a quantitative assessment of the simplified problem can shed light on the more general problem

of transient crustal partial melting and the evolution of the crustal geotherm in response to repeated sill emplacement in arc crust.

2 The Mathematical Model

Following [3], [4], and others, we model the thermal evolution of arc crust by the heat equation

$$\rho C_p \partial_t T + \rho L \partial_t X = \nabla k \nabla T, \tag{1}$$

where T is temperature, t is time, ρ is density, C_p is specific heat capacity, L is specific latent heat, $X \in [0, 1]$ is melt fraction, and k is thermal conductivity (Tables 1 and 2). To model melt fraction X as a function of temperature T, we assume a piecewise linear melting equilibrium relationship (Figure 2b), given by

$$X = \phi_i T + b_i, \tag{2}$$

where ϕ_i and b_i are experimentally determined equilibrium coefficients [3]. The time derivative of (2) is

$$\partial_t X = \phi_i \partial_t T + T \partial_t \phi_i, \tag{3}$$

which we approximate as

$$\partial_t X = \phi_i \partial_t T \tag{4}$$

by assuming $\partial_t \phi_i = 0$, which is true except at discrete temperatures where there are slope breaks (Figure 2b). We come back to this point in Section 3.1. With the approximation in (4), (1) can be expressed as

$$\partial_t T = \nabla k^* \nabla T, \tag{5}$$

where

$$k^* = \frac{k}{\rho(C_p + \phi_i L)}. \tag{6}$$

We solve (5) numerically to find $T(x, y, t),\ x, y \in \Omega,\ t \in [0, \tau]$, where τ is total simulation time, x and y are spatial variables, and Ω is the solution domain. Since X, and thus ∇X, are functions of T, k^* is also a function of T, which introduces nonlinearity to the problem. We assume the following initial and boundary

conditions

$$T(x,y,0) = \begin{cases} T_{sill} & \text{on } \Omega_{sill} = [L_1, L_2] \text{ x } [H_1, H_2], \\ T_0 & \text{elsewhere,} \end{cases} \tag{7}$$

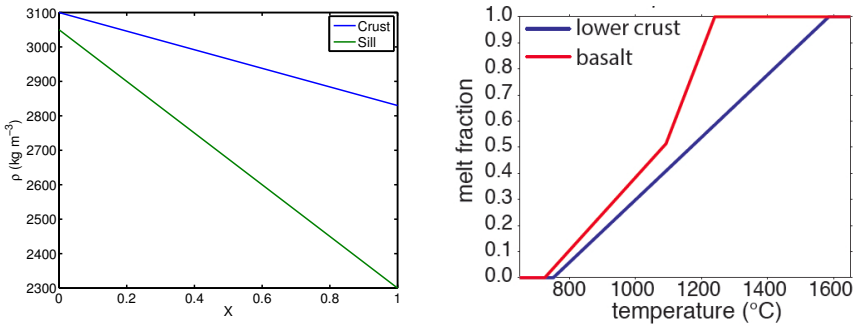
$$\nabla T(x,y,0) = 0 \ \text{ in } \Omega \times [0,\tau], \tag{8}$$

and

$$T(x,y,t)|_{\partial\Omega} = T_0 \text{ on } \partial\Omega \times [0,\tau], \tag{9}$$

where T_{sill} is temperature of intruded sill and T_0 is crustal background temperature. We denote $L_{sill} = L_2 - L_1$ to be sill length and $H_{sill} = H_2 - H_1$ to be sill height.

In all the models we consider, we neglect any melt advection, and we represent the intruded sill in the domain by (7). We assume that the intruded sill is basaltic in composition while the surrounding crust is andesitic in composition (Table 2). We assume that the thermal conductivity k is constant within the sill and crust during the melting and solidification time (i.e, independent of X). We further neglect any pressure dependence on thermal properties. We take the density, ρ to be linearly dependent on melt fraction X (Figure 2a).



(a) (b)

Figure 2: (a) Density mixing relationship; (b) Melt equilibrium assumed in (4).

Table 1: Description of variables used and their units.

Variable	Description	Unit
T	Temperature	K
X	Melt fraction	-
τ_s	Solidification time	years
τ_c	Time duration with crustal $X > 0.2$	years
T_{sill}	Sill temperature	K
T_0	Background temperature	K
ρ	Density	kg m^{-3}
C_p	Specific heat capacity	J kg^{-1}
L	Specific latent heat	J kg^{-1} m^{-1} K^{-1}
k	Thermal conductivity	W m^{-1} K^{-1}
$A_{X>0.2}$	Area with crustal $X > 0.2$	m^2

Table 2: Numerical values of variables used in the models

Parameters		
Symbol [unit]	Domain	Value
ρ [kg m^{-3}]	Ω_{sill} $(X = 0)$	3100
	Ω_{sill} $(X = 1)$	2830
	Ω $(X = 0)$	3050
	Ω $(X = 1)$	2300
C_p [J kg^{-1}]	Ω_{sill}	1480
	Ω	1390
L [J kg^{-1} K^{-1}]	Ω_{sill}	4.0×10^5
	Ω	3.5×10^5
k [J s^{-1} m^{-1} K^{-1}]	Ω_{sill}	2.6
	Ω	2.6
T_{sill} [K]	Ω_{sill}	1558
T_0 [K]	Ω	varies

3 The Finite Element Method

The finite element method is one of the most widely used methods to solve a boundary value partial differential equations on a given domain [13]. We review the essentials of the method here. Generally, the domain is first subdivided into a finite number of subdomains known as finite elements. The solution of the differential equation is then approximated by a set of polynomial functions on each of the elements, where the approximate solution is spanned by a set of orthogonal basis functions in a finite dimensional space V. The finite element solution is an approximate solution $T_{approx} \approx T$ in a subspace $V_h \subset V$ with chosen basis functions $\{\psi_i\}_{i=1}^d = \{\psi_1, \psi_2, ..., \psi_d\}$ such that $T_{approx}(x, y) = \sum_{i=1}^d \xi_i \psi_i(x, y)$, where $d = dim(V_h)$ and ξ_i's are unknown nodal coefficients. Hence, the problem reduces to determine $V_h \subset V$ such that $||T_{approx} - T|| < TOL$, where TOL is the tolerance of error in a specified norm. All of our numerical models in this study employ the standard finite element method.

Since our problem is both time and space dependent, we use Rothe's method to discretize space and time. We first discretize time resulting in a stationary PDE at each time step, which is then solved by standard finite element techniques [1]. The major advantage of using the Rothe's method is the flexibility to use a different finite element mesh at each time step, allowing for adaptive mesh refinement where needed [1]. As such, we solve a time-independent PDE at each time step that can be discretized independently of the mesh used for the previous time steps [1]. Adaptivity of the finite element mesh is used to increase the accuracy of the solution while reducing the computational cost.

3.1 Time Discretization

We discretize time using a finite difference approximation of $\partial_t T$. Let T^n be an approximation at time $t = t_n = n\Delta t$, where $\Delta t = t_n - t_{n-1}$ denotes the length of the present time step. The finite difference approximation of (1) is

$$\frac{T^n - T^{n-1}}{\Delta t} - \nabla k^{*n} \nabla [\theta T^n + (1-\theta) T^{n-1}] = 0, \tag{10}$$

on Ω, where $\theta \in [0, 1]$ is a constant,

$$k^{*n} = \frac{k}{\rho(C_p + \phi_i^{n-1} L)}, \tag{11}$$

and ϕ_i^{n-1} is the equilibrium coefficient in the previous time step. The boundary condition is $T^n = T_0$ on $\partial\Omega$.

When $\theta = 0$, (10) reduces to the forward, or explicit, Euler Method. When $\theta = 1$, it reduces to the backward, or implicit, Euler method. Both explicit and implicit Euler methods yield first order accurate solutions. If $\theta = 0.5$, then the semi-discretized equation (10) results in the Crank-Nicholson method which yields a second order accurate solution. We include θ as one of the user defined parameters in our developed program; however, since the PDE in (1) is parabolic, we use $\theta = 1$. For parabolic PDEs, the implicit method guarantees unconditional stability [14].

One of the major challenges in solving the problem described in Section 2 is the presence of the nonlinear variable k^*, which is dependent on T through (4). In order to compute the solution at the present time step, we use the solution at the previous time step which allows us to solve the time dependent problem trivially. Since we assume piecewise linear melting equilibrium equations, ϕ_i is constant except at discrete times

(Figure 2b). However, this technique is inaccurate unless the time step is very small and the equilibrium equation is chosen such that the phase change occurs over sufficiently large temperature interval [7]. In view of this restriction, we minimize the time step size to be between 0.01 and 0.1 year for each of the simulations in this study, which results in small change in temperatures between successive time steps.

3.2 Space Discretization

We discretize space using the standard finite element method [8]. We find the weak form of (1) by first multiplying with a test function from the left, and integrating over the entire domain, integrating by parts wherever necessary. We use the discrete approximation of the solution at the n^{th} time step, $T_h^n \in V_h$, where V_h is the finite dimensional subspace of the Sobolev space V, and solve

$$(T_h^0, \varphi_h) = \begin{cases} (T_{sill}, \varphi_h) & \text{on } \Omega_{sill} = [L_1, L_2] \text{ x } [H_1, H_2], \\ (T_0, \varphi_h) & \text{elsewhere,} \end{cases} \tag{12}$$

$\forall\ \varphi_h \in V_h$ for $n = 0$, where φ_h is the basis of V_h, $(u, v) = \int_\Omega uv\, d\Omega$ is the L^2 inner product, T_h^0 is the approximate solution at $t = 0$, and

$$(T_h^n, \varphi_h) + \frac{k}{\rho(C_p + \phi_i^{n-1}L)}\Delta t\ \theta(\nabla T_h^n, \nabla\varphi_h) = (T_h^{n-1}, \varphi_h) - \frac{k}{\rho(C_p + \phi_i^{n-1}L)}\Delta t\ (1-\theta)(\nabla T_h^{n-1}, \nabla\varphi_h) \tag{13}$$

$\forall\ \varphi_h \in V_h$ for $n \geq 1$. Using the standard Galerkin approximation [1], we approximate $T^n(x, y)$ at the n^{th} time step as

$$T^n(x, y) \approx T_h^n(x, y) = \sum_i \xi_i^n(x, y)\ \psi_i^n(x, y) \tag{14}$$

where T_h^n are trial functions, $\psi_i^n(x, y)$ are shape functions used at the n^{th} time step and ξ_i^n are unknown temperature solutions at the nodes.

Substituting approximation (14) into (13), we reach the following equation

$$\left(M^n + \frac{k}{\rho(C_p + \phi_i^{n-1}L)}\Delta t\ \theta A^n\right)\xi^n = M^{n,n-1}\xi^{n-1} - \frac{k}{\rho(C_p + \phi_i^{n-1}L)}\Delta t\ (1-\theta)A^{n,n-1}\xi^{n-1}, \tag{15}$$

where

$$M^n_{i,j} = (\psi^n_i, \psi^n_j),\ M^{n,n-1}_{i,j} = (\psi^n_i, \psi^{n-1}_j),$$

$$A^n_{i,j} = (\nabla\psi^n_i, \nabla\psi^n_j),\ A^{n,n-1}_{i,j} = (\nabla\psi^n_i, \nabla\psi^{n-1}_j).$$

Lastly (15) is recast into a more convenient form by dividing both sides by $\Delta t > 0$, to yield

$$\left(\frac{1}{\Delta t}M^n + \frac{k}{\rho(C_p + \phi_i^{n-1}L)}\theta A^n\right)\xi^n = \left[\frac{1}{\Delta t}M^{n,n-1} - \frac{k}{\rho(C_p + \phi_i^{n-1}L)}(1-\theta)A^{n,n-1}\right]\xi^{n-1}, \tag{16}$$

where M^n and A^n in (16) are commonly referred as the mass matrix and stiffness matrix respectively. Solving (16) at the current time step allows us to advance to the next time step using the finite difference approximation in (10) to achieve time dependent solutions of T and X. Note that since the solution of the previous step may have been computed on a different mesh, we use the shape functions $\psi_i^{n-1}(x, y)$ from the previous time step to map T_h^{n-1} onto the current mesh [1].

4 Implementation and Mesh Refinement

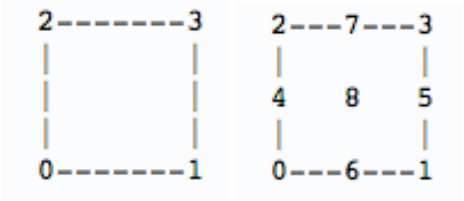
Our finite element program employs object-oriented programming to organize data and functions via the use of classes. One of the advantages of this practice is that we can compose finite element implementations into smaller blocks. All user defined inputs are read at the beginning of the program from the input file called `parameter.prm`.

In Section 3.2, we approximate the solution to the problem described in Section 2 at the n^{th} time step by

$$T_h^n(x, y) = \sum_i \xi_i^n(x, y)\ \psi_i^n(x, y), \tag{17}$$

where $T_h^n(x, y) \in V_h$, ψ_i^n's are shape functions at the n^{th} time step, and $\xi_i^n(x, y)$'s are unknown temperature nodal solutions. The total number of nodal temperatures is referred to as the degree of freedom, DoF, of the problem. In order to solve for the temperatures, the finite dimensional function $T_h^n(x, y) \in V_h$ in (17) needs to satisfy the weak formulation given in (15) for all test functions in V_h. To do so, we numerically integrate all the integral terms that appear in (15) by using Gauss-Legendre quadrature formula on a mesh [1].

We choose our subspace V_h to be the scalar Lagrangian space to obtain a finite element space of contin-



(a) (b)

Figure 3: (a) Numbering of nodes for a linear element, which consists of one cell; (b) Numbering of nodes for a quadratic element, which consists of four cells [1].

uous, piecewise polynomials of degree 2 in each coordinate direction [1]. In other words, we use quadratic or Q2 finite elements, with each element composed of 9 nodes.

In `deal.II`, the finite element mesh is considered to be composed of cells. A cell is a geometrical image in $\mathbb{R}^d$ connecting two or more nodes. In 1D, a cell is simply a line and in 2D it is a quad. In contrast, a finite element is a subdomain of the solution domain, and a cell is equivalent to an element only for linear (Q1) elements (Figure 3a). In 2D, each quadratic (Q2) element is composed of four cells (Figure 3b). Each of the cells represents crust material. Figure 3 illustrates the distinction between a cell and an element as well as the node numbering for Q1 and Q2 elements in 2D.

In the numerical computation of the problem, we aim to increase the spatial resolution where the solution is less regular. We do this by implementing a scheme to locally refine (i.e. increasing the grid points) at the interface between the emplaced sill and the surrounding crustal area. We achieve this goal via local adaptive mesh refinement based on error estimator[1] proposed by Kelly et al. (1983). The error estimator is implemented as the function `KellyErrorEstimator` within the `deal.II` library. This estimator approximates the error of the finite element solution in each element by integration of the jump of the gradient of the solution along the faces of each cell [10]. In other words, the error estimator assigns to each cell K the indicator

$$\eta_K = \left(\frac{h_K}{24} \int_{\partial K} \left[\frac{\partial T_h}{\partial n} \right]^2 d\sigma \right)^{1/2}, \tag{18}$$

where h_K is greatest length of the diagonals of cell K and $[\cdot]$ denotes the jump of the normal derivatives across a face $\gamma \subset \partial K$ of the cell K [1], [10]. It is used to refine the regions of the mesh with the largest $R\%$ error and coarsen the regions of the mesh with the smallest $C\%$ error, where R and C are user specified refinement parameters.

[1]Note that it is not an a posteriori error estimator. It merely acts as an indicator for mesh refinement [1].

We implement h-refinement in our finite element program. With h-refinement, the mesh connectivity changes as the mesh is refined. All of the meshes used in this study are structured, square grids. For each level of adaptive refinement, a marked cell is isotropically subdivided into four equivalent square child cells, resulting in a change in the overall number of nodal points, but that preserves the topology of the overall mesh [2]. The finer mesh overlies the coarser one, and the DoF increases after each level of refinement. This results in a new, more localized mesh after the marked cells are refined. The adaptivity strategy in `deal.II` uses grids in which neighboring cells are refined independently, which can result in additional nodes on the interfaces of cells which belong to one side but not the other side (Figure 5). These so-called hanging nodes are constrained in order to guarantee the continuity of solution [1]. The constraint on hanging nodes is such that the temperature on the element face containing a hanging node is linear [1].

One particular challenge to obtain an accurate solution is the interpolation and projection of the problem geometry and initial conditions on the initial mesh at $t = 0$. Specifically, the temperature jump between the sill and surrounding crust is difficult to resolve since the initial temperature field is not differentiable at the sill-crust interface. We compute the L^2 projection[2] of the initial temperature field onto our discretized finite element mesh, which results in a smooth projection of the initial temperatures onto the initial mesh. The smoothness of the projected initial temperatures technically violates our initial condition of a finite jump in temperature at the sill-crust interface. However, we minimize the difference between the idealized and applied initial conditions by adaptively refining the initial mesh a few times so that an increased density of cells around the sill-crust interface more closely approximates the idealized initial conditions (Figure 9). This initial refinement increases our computational cost and we choose the optimal number of pre-refinement steps on the initial mesh via trial and error.

In `deal.II`, the linear finite element system is assembled at each time step by computing the contribution of each cell in a small matrix, and then transferring these to the global matrix [1]. The k^* at the present time step is computed based on the solution in the last time step (see Section 3.2). The linear system, which is almost symmetric and positive definite, is solved using a conjugate gradient solver with SSOR preconditioner [1], obtaining the numerical solution at the current time step. The solution is updated through time using the finite difference approximation as described in Section 3.1. The mesh is adaptively refined between time steps based on user specifications, and the solution is mapped onto the new mesh when refined. We do this

[2]The L^2 projection $P_h u \in V_h$ of a function $u \in L_2(\Omega)$ into the space V_h on a triangulation of a domain Ω is defined as $(u - P_h u, v) = 0 \; \forall \; v \in V_h$ [13].

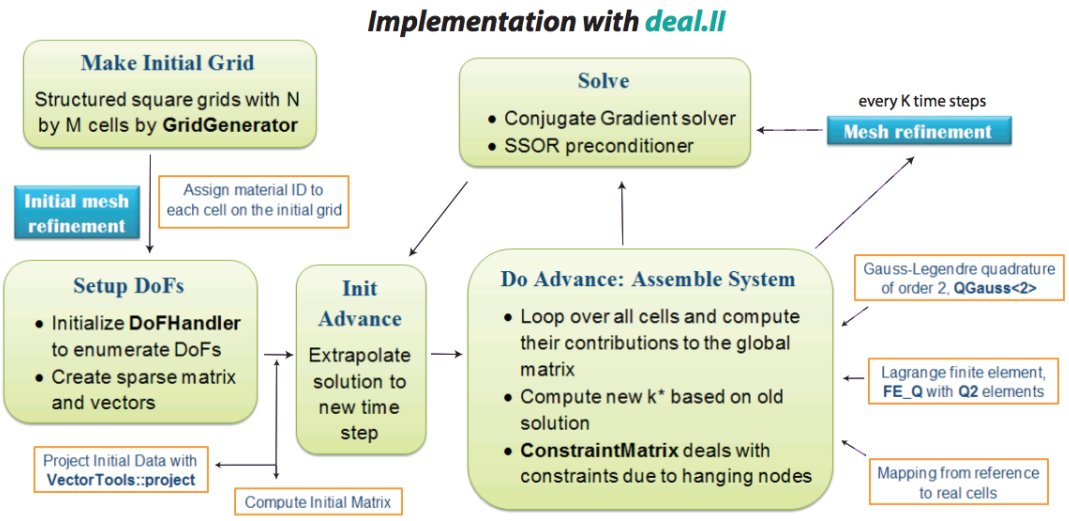


Figure 4: Flow chart summarizing the implementation of the finite element program.

via the function `SolutionTransfer` in `deal.II`, and regenerate the system matrix after each refinement [1]. However, it is worthwhile to note that this approach introduces unavoidable interpolation error each time we adaptively refine the mesh [1]. We stop the simulation when the entire domain is completely solidified, i.e. when melt fraction $X = 0$ for all $(x, y) \in \Omega$ and we denote the final time as the solidification time, τ_s. Figure 4 shows a flow chart of the organization and steps of our program. We recommend interested readers to refer to [1] for complete discussion of related implementation details.

Our program postprocesses the computed temperature solutions using `Gnuplot` for visualization. We use the obtained temperature solutions to calculate the corresponding melt fraction in the sill and crustal domain. Another quantity of interest that we compute is the area with crustal melt fraction above 0.2, denoted as A or $A_{>0.2}$ (see Section 6) and we do so by using bilinear interpolation of the melt fraction solutions to determine if the associated area of the cells on the mesh contributes to $A_{X>0.2}$ (Figure 6).

5 Validations

We first validate our finite element program using two cases in which an analytical solution is known. These cases are heat conduction following an intrusion of a 500 m $\times$ 50 m rectangular sill of temperature $T_{sill} = 1558$ K into the 2.5 km $\times$ 1.0 km crustal domain at time, $t = 0$. The intruded sill occupies the subdomain $[L_1, L_2] \times [H_1, H_2]$ and we choose this subdomain such that the sill is intruded at the center of

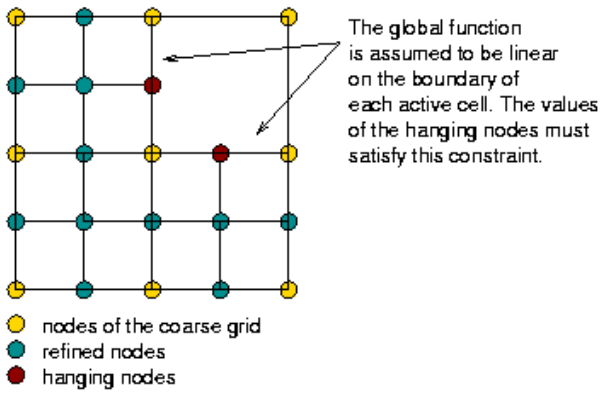


Figure 5: Illustration of hanging nodes, resulting from mesh refinement for Q1 finite elements (from [1]).

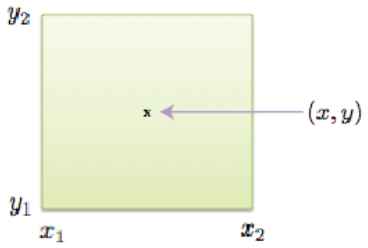


Figure 6: Computation of area of a cell with associated crustal melt fraction above 0.2, $A_{X>0.2}$. Let $x = \frac{x_1+x_2}{2}$, $y = \frac{y_1+y_2}{2}$. For the crustal area of $(x_2 - x_1)(y_2 - y_1)$ to contribute to $A_{X>0.2}$, the criterion that needs to be satisfied is: interpolated melt solution at (x, y), $X_{interp} = (1-x)(1-y)X(x_1, y_1) + x(1-y)X(x_2, y_1) + y(1-x)X(x_1, y_2) + xyX(x_2, y_2)) \geq 0.2$, where $X(x_i, y_i)$ is the melt fraction solution obtained at (x_i, y_i).

the domain. We assume that both the crustal domain and the intruded sill share the same coefficient of thermal conductivity $k^* = k/C_p\rho$ with $\phi_i = 0$. This assumption eliminates any nonlinearity introduced in the governing PDE as there is no temperature dependence in k^*. In other words, we assume $\nabla X = 0$ so that $k^* = \frac{k}{C_p\rho}$ is a constant over the entire domain. We choose $\Omega = [0, L] \times [0, H] = [0, 2.5] \times [0, 1.0]$, $dt = 0.1$ years, and the following boundary and initial conditions

$$\frac{\partial T(x, y, t)}{\partial y}|_{\partial\Omega} = g_0, \tag{19}$$

$$\frac{\partial T(x, y, t)}{\partial x}|_{\partial\Omega} = 0, \tag{20}$$

$$T(x,y,0) = \begin{cases} T_{sill} = 1558 \text{ K} & \text{on } [1.0, 1.5] \text{ x } [0.475, 0.525], \\ T_0 - \frac{g_0 H}{2} + g_0 y & \text{elsewhere,} \end{cases} \tag{21}$$

such that $T(x, H/2, 0) = T_0$, where g_0 represents a linear geotherm. We take $g_0 = 0$ K/m and $g_0 = 30$ K/m for the first and the second case respectively, and refer to these two cases as validation model 1 and validation model 2, respectively. Our goal in these validation models is to demonstrate that over the domain assumed, the geotherm g_0 has negligible effect on the temperature solution, which justifies our assumption that the background temperature in the crustal region is a constant (see Section 6). The boundary conditions in (19) and (20) imply that $T(x, 0, t) = T_0 - g_0 H/2$, $T(x, H, t) = T_0 + g_0 H/2$, and $T_y(0, y, t) = T_y(L, y, t) = g_0$. See Appendix A for the derivation of the analytical solution.

To solve both validation problems, we choose the initial mesh to consist of 10×10 cells and refine the initial mesh adaptively 7 times. At $t > 0$, we allow adaptive refinement every 50 time steps and stop the program after 50 years (500 time steps). Adaptive mesh refinement in both problems are performed by using the error estimator `KellyErrorEstimator`. We specify the program to refine the regions with the highest 30% error and coarsen the regions with the lowest 10% error for the initial adaptive refinement and to refine the regions with the highest 0.5% error and coarsen the regions with the lowest 0.25% error for adaptive refinement at later times (Figure 10). We evaluate the accuracy of the numerical solutions to both validation models by comparing them to the analytical solutions. The analytical solution is expressed as an infinite series which decays in time, and we compute only the first 500 terms of the series (Figure 7 and 8). There is a negligible error associated with the truncated series solution.

Figure 11 shows the numerical solution at four specified locations near the sill-crust interface for validation problem 1. Figure 12 and Figure 15 show the numerical solution at four specified locations near the sill-crust interface and the relative error, respectively for validation problem 2. With validation problem 1, we further compare the performance yielded by different mesh refinement strategies, namely adaptive refinement with `KellyErrorEstimator` as error indicator vs. adaptive refinement with `DerivativeApproximation` as error indicator vs. a global refinement of the initial mesh (15×15 cells) three times without adaptive remeshing at later times, where `DerivativeApproximation` is an error estimator function available within `deal.II` and is based on the finite difference approximation to the second derivative of the cells [1]. The error estimator function `DerivativeApproximation` assigns to each cell K the indicator

$$\eta_K = h_K^3 \|\nabla_h T_h(K)\|_\infty, \tag{22}$$

where h_K is greatest length of the diagonals of cell K and $T_h(K)$ denotes T evaluated at the center of the cells [1]. Comparing the performance of these three refinement strategies allows us to determine the optimal strategy for numerical modeling in Section 6.

Figure 13 compares the nodal temperatures using the three refinement strategies, while Figure 14 illustrates the relative error yielded by using those strategies. The relative error is computed as

$$e = \frac{T_c - T_a}{T_a} \times 100\%, \tag{23}$$

where T_a are the temperatures from the analytical solution and T_c are from the numerical solution. We observe that the three refinement strategies yield somewhat equally accurate solutions over time, but the first and second refinement strategies perform better than the global refinement strategy in projecting the initial conditions onto the mesh. In terms of DoFs required by these three strategies, we note that although the initial global refinement strategy yields equally accurate solution as the first and second strategy, the DoFs (and hence computational cost) required when using this strategy is much higher. On the other hand, the first and second strategies require roughly equal DoFs, with DoFs of the first strategy slightly lower than the second strategy. In view of this analysis, we employ the first strategy for mesh refinement in the numerical models presented in Section 6.

For both validation problems, we observe that the temperatures in the crustal region near the sill-crust interface increase rapidly in the first decade and decay exponentially thereafter. The relative error of the numerical solutions for both cases decreases rapidly from roughly 10% (due to projection of initial conditions onto the mesh) at initial time to less than 0.1% in about three years time. Note that the reason we only choose four specified points near the sill-crust interface to perform error analysis but not evaluate the error norms[3] is that local pointwise error analysis yields more meaningful analysis for our problems since we are mainly interested in melt fraction near the sill-crust interface. In Figure 16, we confirm that the difference in solutions to the two validation problems is bounded above by 0.1%, justifying our assumption that the geotherm in the domain can be neglected in the models presented in Section 6.

[3]In fact, evaluating the error norms is computationally costly due to the huge number of cells on the mesh after the initial adaptive refinements.

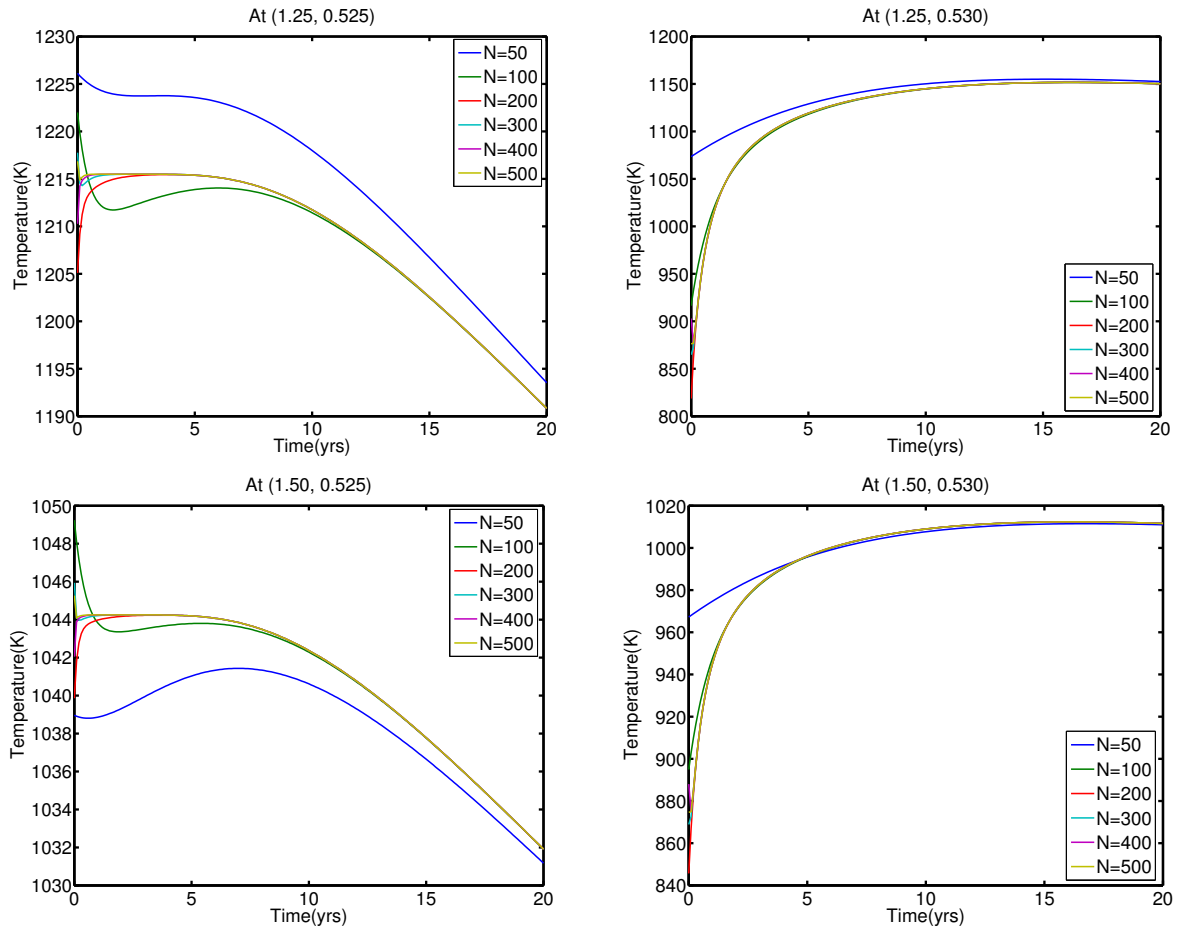


Figure 7: Validation model 1: Convergence of series solution with increasing number of terms, N.

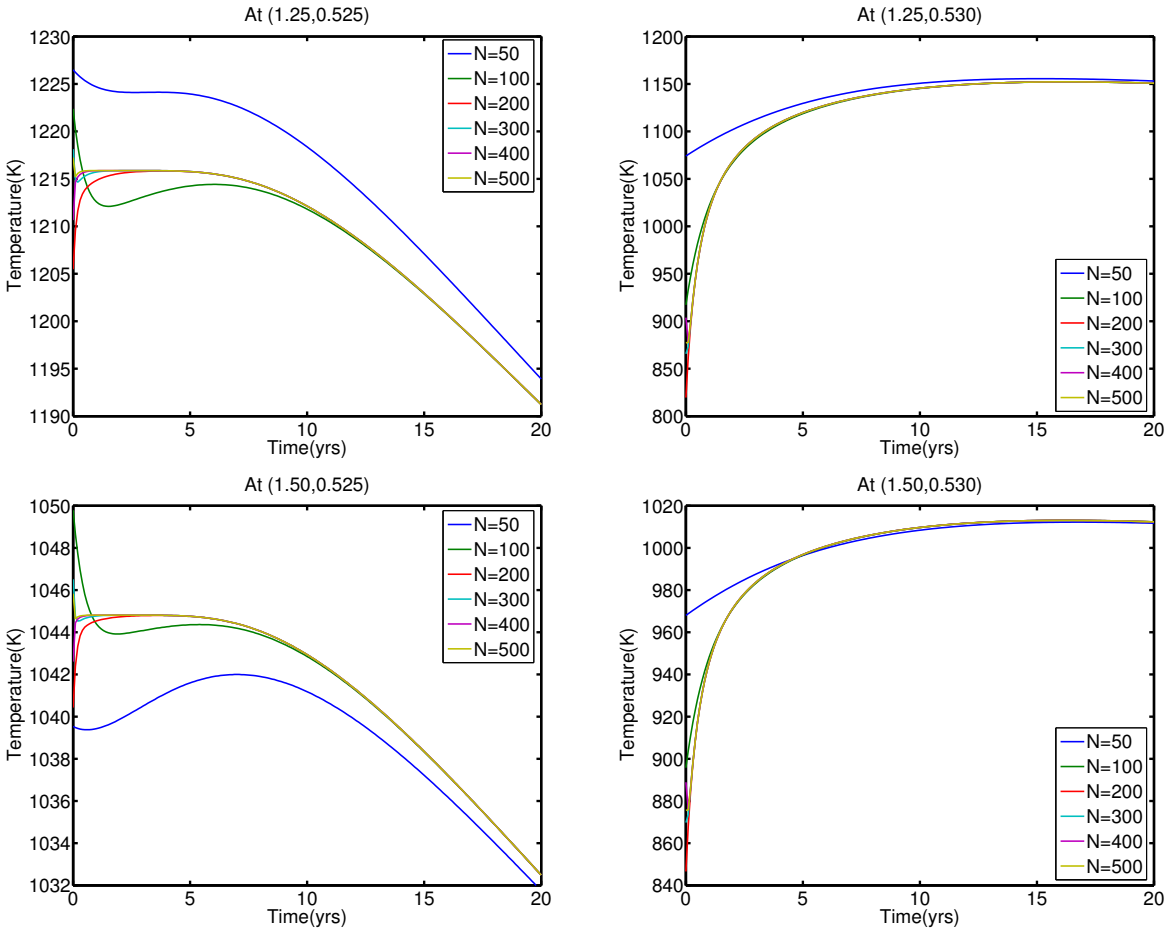


Figure 8: Validation model 2: Convergence of series solution with increasing number of terms, N.

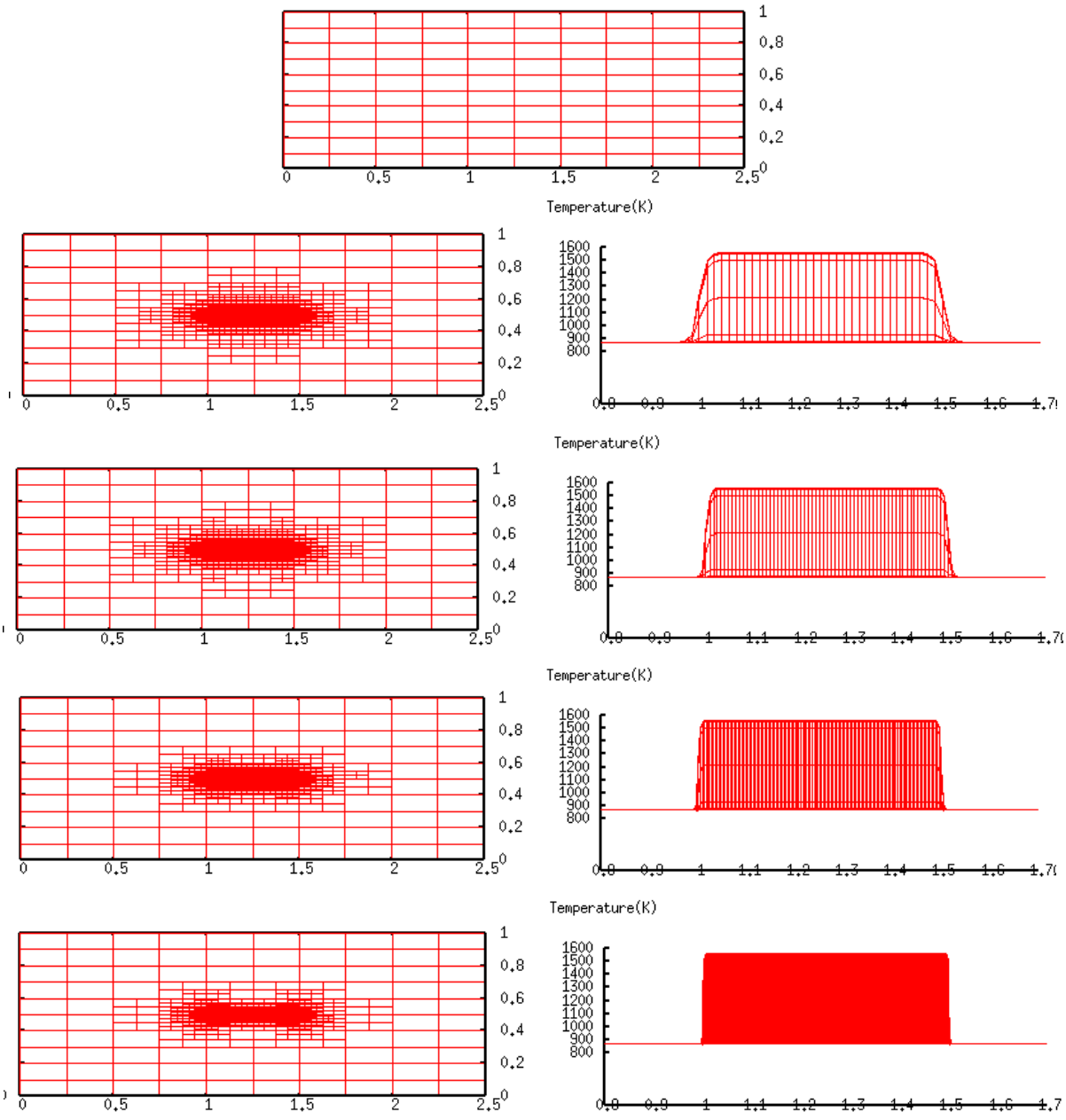


Figure 9: Validation model 1: Projected initial conditions on the mesh after K times of pre-adaptive refinement with KellyErrorEstimator as error indicator. Left: Adaptive mesh with $K = 4$, $K = 5$, $K = 6$ and $K = 7$; Right: Projected initial conditions on the mesh with $K = 4$, $K = 5$, $K = 6$ and $K = 7$. Observe that the projection on the mesh tends to the idealized projection as K increases.

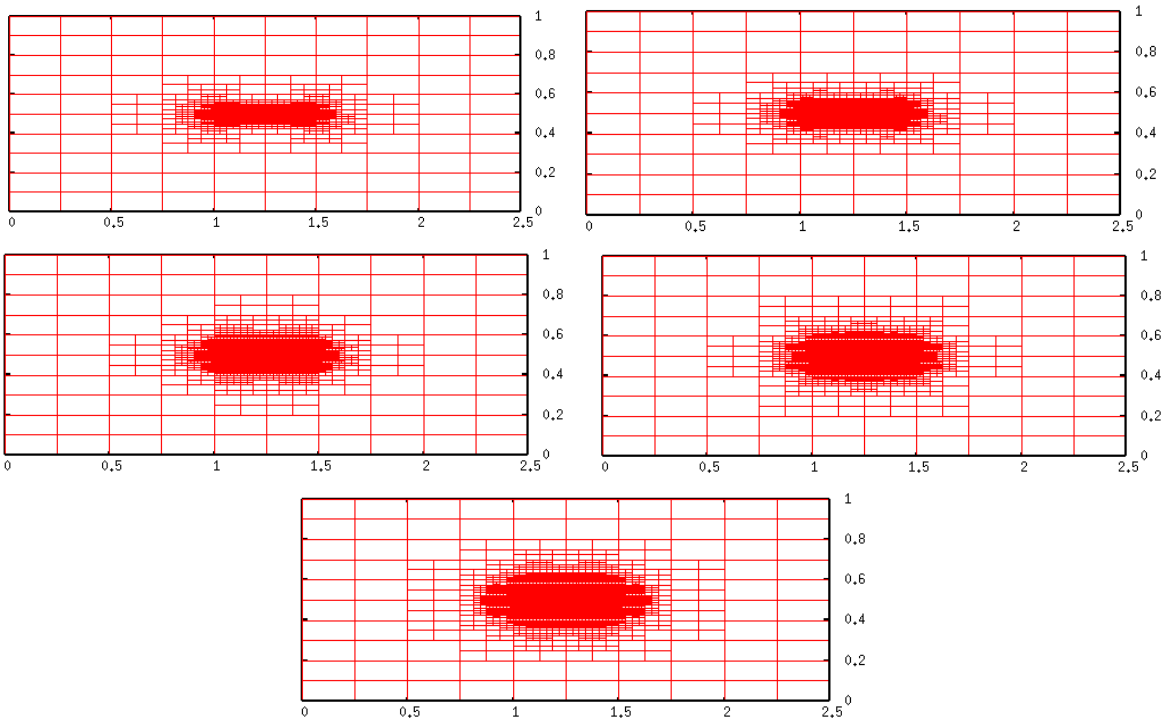


Figure 10: Validation model 1: Adaptive meshes using KellyErrorEstimator as error indicator every 50 time steps. From top left to top right to bottom: Adaptive mesh obtained at t = 0, 10, 20, 30 and 40 years.

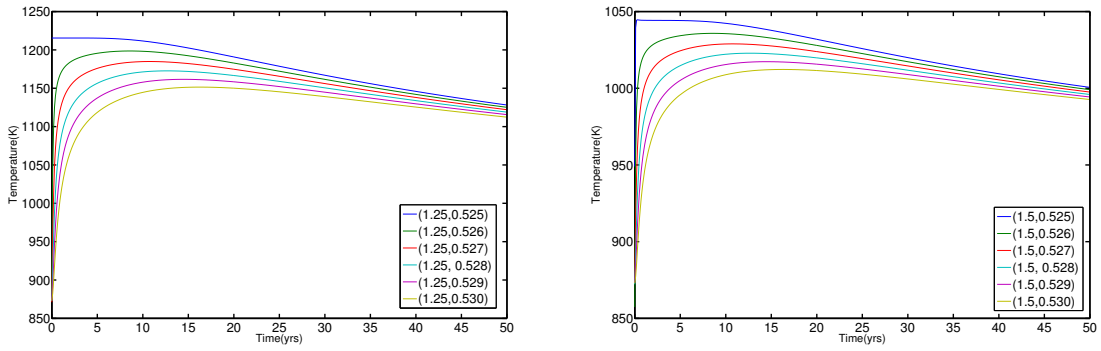


Figure 11: Validation model 1: Numerical solutions at specified locations using KellyErrorEstimator over time.

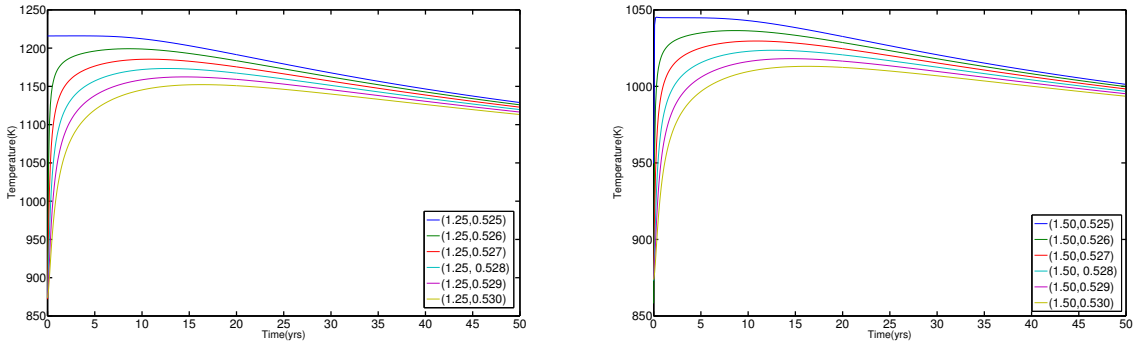


Figure 12: Validation model 2: Numerical solutions at specified locations using KellyErrorEstimator over time.

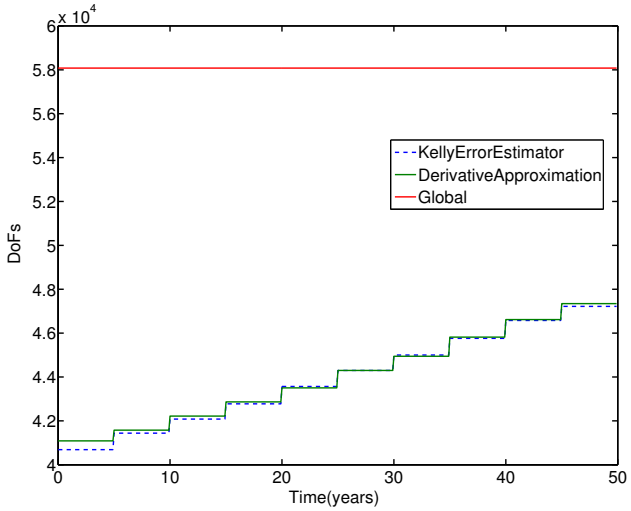


Figure 13: Validation model 1: Comparison of DoFs over time using different refinement strategies. Here the DoFs is on order of 10^4.

6 Numerical Models of Partial Transient Melting

We consider 16 models with rectangular sills of initial $T_{sill} = 1558$ K, initially placed into the domain of length L and height H with lower crustal composition and initial background temperature of $T_0 = 873 - 973$ K. We consider H and L to be between 0.2 - 4.0 km and 2.5 - 20 km, respectively, where the exact size depends on the sill size and background temperature. That we assume a constant background temperature surrounding the intruded sill is justified in Section 5. We define τ_c and A (or $A_{X>0.2}$) as the time duration and area, respectively, with crustal melt fractions above 0.2. We consider $X > 0.2$, since this level of melt fraction may result in melt migration and segregation in the lower crust [9]. It is worth noting that all the models considered here are local in nature and that we have neglected external forcings and interactions that

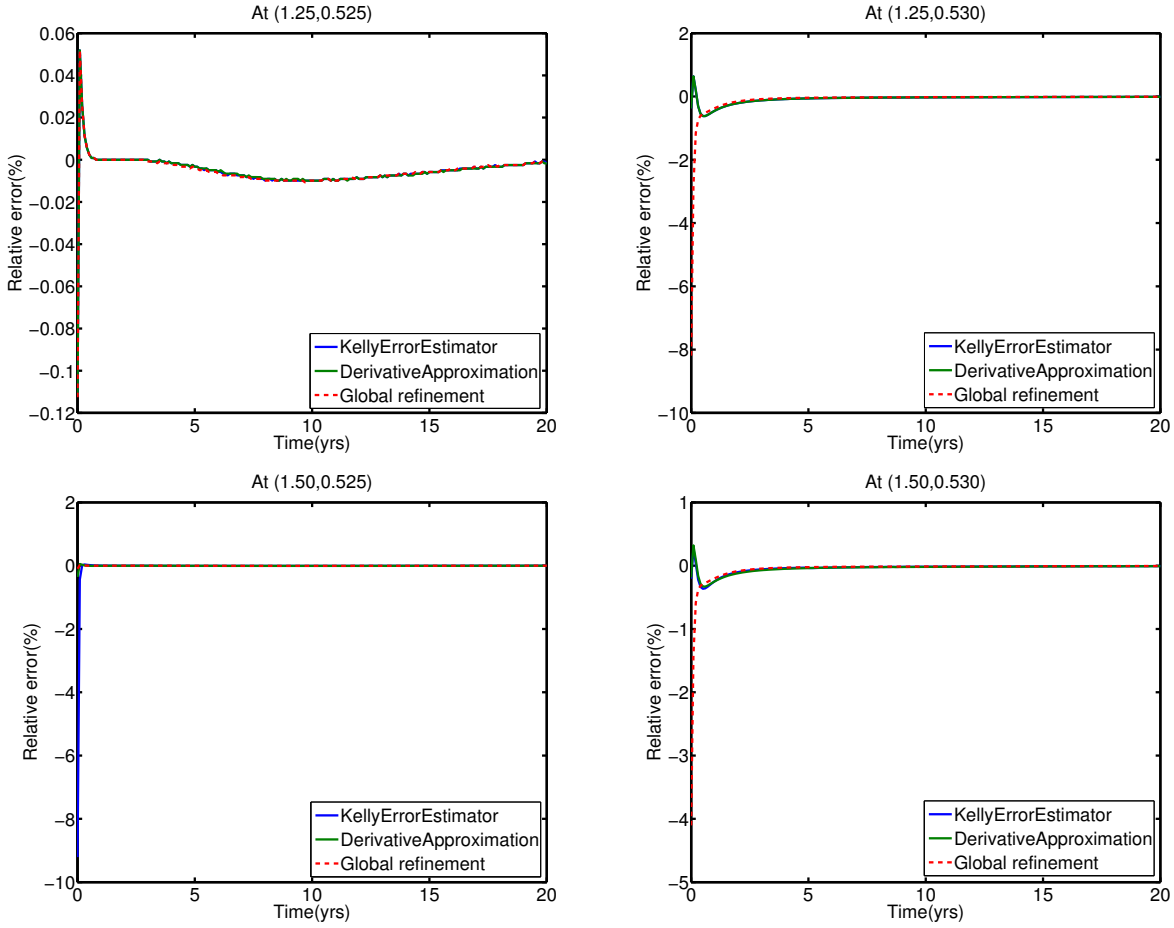


Figure 14: Validation model 1: Comparison of relative error obtained using different refinement strategies.

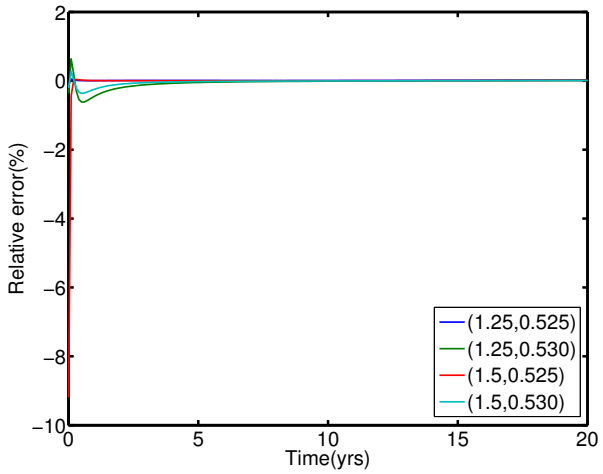


Figure 15: Validation model 2: Relative error of numerical solutions over time at specified locations.

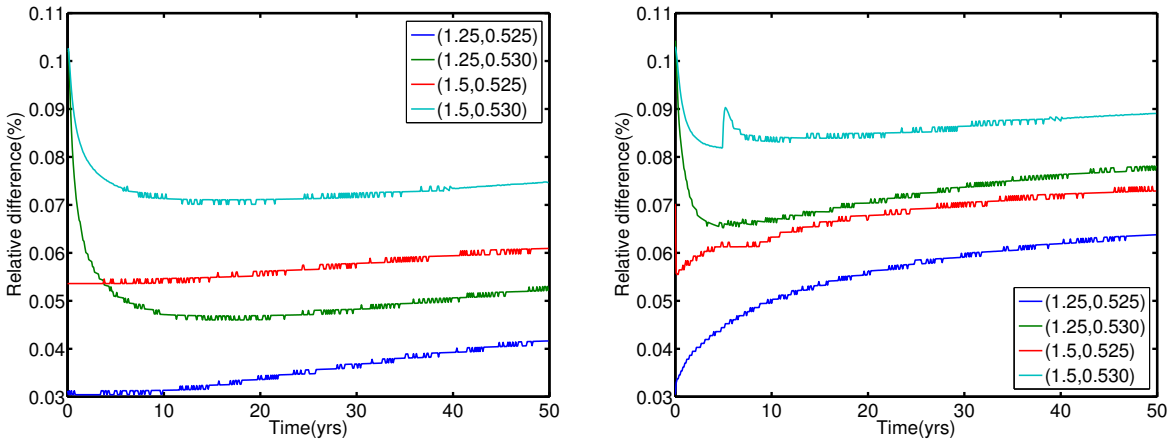


Figure 16: Left: Relative difference between analytical solutions to validation model 1 and 2 over time at specified locations; Right: Relative difference between numerical solutions (with KellyErrorEstimator as error indicator for refinement) to validation model 1 and 2 over time at specified locations. Relative difference is computed as the absolute value of the difference in solutions over the solution to validation model 2 multiplied by 100%.

Table 3: Models with sill height H_{sill}= 0.05 km at T_0 = 873 K.

Model	L_{sill} (km)	τ_s (yrs)	τ_c (yrs)	Max. $A_{X>0.2}$ (m^2)
1	0.05	57.48	0.49	8.09
2	0.1	111.04	1.54	35.2
3	0.5	310.2	10.3	370.5
4	1.0	318.1	19.2	843.2
5	5.0	317.9	47.2	19950

could contribute to the process. Tables 3 - 6 summarize the results of the models. Figure 17 shows the cost of performing the major operations in our finite element program for model 4.

We find that both initial crustal temperature and dimensions of the sills are important parameters for the transient partial melting process. Following the intrusion of a 1 km by 50 m sill, the surrounding lower crust partially melts, with melt fractions greater than 0.2 persisting over an area up to 843 m^2 and 11015 m^2 for a crust with background temperature of 873 K and 973 K, respectively. If the partial crustal melt at time of maximum melting migrated to a secondary sill of the same length, this area would correspond to secondary sill of at least $\sim$ 1 m or 3 m if originating in 873 K or 973 K respectively.

Although the highest amount of the crustal melt is generated early after sill emplacement, the complete partial melting process can occur over a long time period, depending on the parameter values chosen (Figure 19). For instance, the crustal melt generated by the 5 km long intruded sill in Model 5 and Model

Table 4: Models with sill height $H_{sill} = 0.05$ km at $T_0 = 973$ K.

Model	L_{sill} (km)	τ_s (yrs)	τ_c (yrs)	Max. $A_{X>0.2}$ (m^2)
6	0.05	238.67	11.49	241.66
7	0.1	473.8	27.2	486.09
8	0.5	2113.3	53.7	5107.1
9	1.0	3436.7	58.4	11014.7
10	5.0	5281	149	94742

Table 5: Models with sill length $L_{sill} = 1.0$ km at $T_0 = 873$ K.

Model	H_{sill} (km)	τ_s (yrs)	τ_c (yrs)	Max. $A_{X>0.2}$ (m^2)
11	0.01	12.79	1.45	122.8
12	0.02	50.94	4.46	278.5
13	0.1	1239	31.3	1412

10 decreases at a slower rate towards complete solidification compared to the models where the length of the intruded sill is shorter, demonstrating high persistence of crustal melting over time. However, it takes 318 years and 5281 years for the crust in Model 5 and Model 10 to completely solidify, respectively. The fact that τ_s approaches an asymptote with increasing L_{sill} in Figure 21 supports this observation. Figure 20 shows the effect of varying background temperature on max. $A_{X>0.2}$. Figure 23 show that sills of longer length melt the surrounding crust to a larger degree, while Figure 24 demonstrates the evolution of melts generated by an intruded sill in a crust of background temperature of 873 K vs. that of 973 K over time.

High crustal melt fractions persist for decades following emplacement of a sill, with the time scale at which the melt persisting depending on the sill dimension and initial crustal temperature (Figure 22). At initial crustal temperature of 873 K, a melt fraction as high as 0.2 can persists up to 0.49 years for a 0.05 km by 0.05 km sill and up to 47.2 years for a 1 km by 0.05 km sill. The crustal area with melt fraction of up to 0.2 also increases with the initial crustal temperature. The relationship of max. $A_{X>0.2}$ and initial crustal crustal temperature appears to follow a power law.

The relationship of both max. $A_{X>0.2}$ and τ_c to L_{sill} can be described by a power law (Figure 25). The power laws deduced are max. $A_{X>0.2} \sim L_{sill}^{1.6356}$ and max. $A_{X>0.2} \sim L_{sill}^{1.3144}$ for the case with background

Table 6: Models with sill length $L_{sill} = 1.0$ km and sill height $H_{sill} = 0.05$ km, 873 K $< T_0 <$ 973 K.

Model	T_0 (K)	τ_s (yrs)	τ_c (yrs)	Max. $A_{X>0.2}$ (m^2)
14	898	425.9	16.8	534.7
15	923	621.9	29.9	1455.5
16	948	1009	41.2	3023.7

Total wallclock time elapsed since start	3.05e+05s		
Section	no. calls	wall time	% of total
Assembling system	3107	1.77e+04s	5.8%
Rebuilding matrix	3115	9.11e+04s	30%
Solve system	3107	1.89e+05s	62%
Remeshing	17	132s	0.043%
Setup dof systems	1	0.252s	8.3e-05%

Figure 17: Computational cost for Model 4.

temperature of 873 K and 973 K, respectively. Additionally, $\tau_c \sim L_{sill}^{0.9991}$ and $\tau_c \sim L_{sill}^{0.5070}$ for the case with background temperature of 873 K and 973 K, respectively. The relationships of τ_c, τ_s, and max. $A_{X>0.2}$ to H_{sill} for the case with background temperature of 873 K can also be described by a power law (Figure 26): max. $A_{X>0.2} \sim H_{sill}^{1.0809}$, $\tau_c \sim H_{sill}^{1.3696}$, and $\tau_s \sim H_{sill}^{1.9880}$. Similarly, the relationship of τ_s and L_{sill}, as well as that of τ_c, τ_s, and max. $A_{X>0.2}$ to T_0, seem to obey a power law (Figure 27).

Our results that the amount of crustal melting generally increases with sill dimension and initial crustal temperature suggests that segregation and migration of melt is more important for larger sills and as the geotherm increases due to continued intrusion. While these results yield some insights about the effect of transient partial melting from a single sill, the actual scenario in the lower crust is much more complicated due to the repetitive intrusion of sills over time. However, the duration between emplacement of successive sills is on order of 10^4 years [3], [4], which is much longer than the duration of these simulations. Intuitively, more emplaced sills into the crust raises background temperature, and thus later sills will generate more crustal melt. This may result in migration and segregation of the crustal partial melt and subsequent cooling of the lower crust and heating of the upper crust [5]. For completeness, we need to consider the geometry of the sills, the randomness of the intrusion events, and the orientations in which the sills are intruded in the lower crust [4].

7 Conclusions

We have developed a finite element program based on the `deal.II` library to solve for the time dependent transient partial melting problem following the intrusion of a single sill into the lower crust with a constant background temperature. We employ the continuous Galerkin method with h-adaptive refinement in a structured mesh and an iterative approach to obtain the numerical solutions of both temperature and melt fraction.

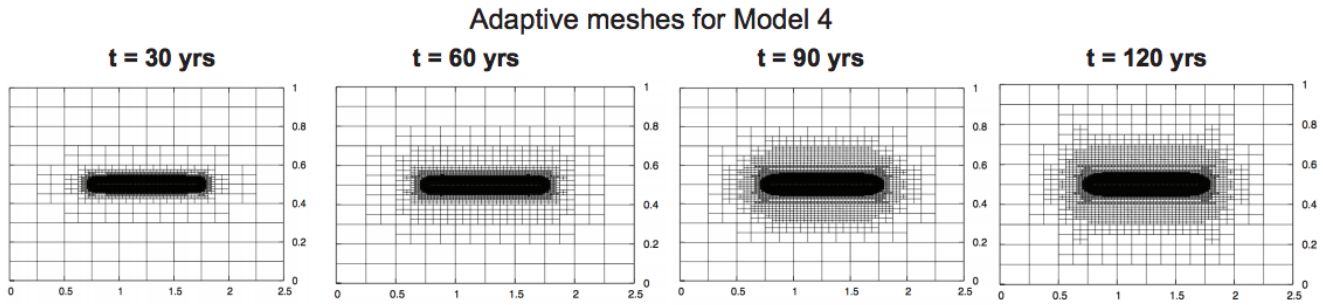


Figure 18: Adaptive meshes for Model 4 at t = 30, 60, 90, and 120 year.

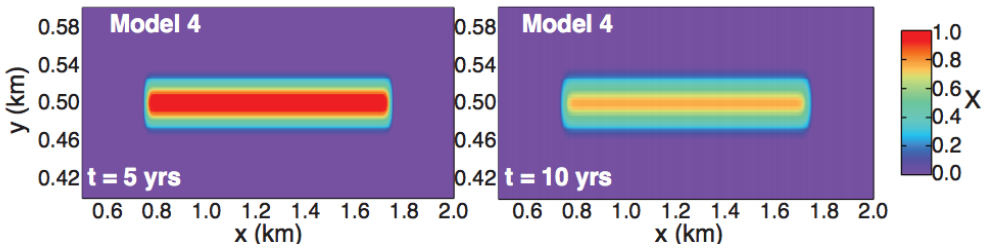


Figure 19: The melt fraction solutions for Model 4 at t = 5 year and t = 10 year. Note that the melt fraction decreases significantly during the period of 5 years.

We find that initial crustal temperature and the dimensions of the sills are the two primary parameters controlling the production of crustal melt. The amount of crustal melting increases with both sill dimension and initial crustal temperature. A higher portion of the lower crust is partially molten and sustain melt fraction higher than 0.2 when we increase the value of both parameters. As a result, migration and segregation of melt may occur due to the high crustal melt productivity generated by the intrusion event. We further deduce that the relationship of τ_c and L_{sill} as well as that of max. $A_{X>0.2}$ and L_{sill} appear to follow power law relationships. The future direction would be to extend the single sill intrusion problem to multiple and repetitive emplacement of sills.

8 References

[1] Bangerth, W. , Heister, T., & Kanschat, G. `deal.II` differential equations analysis library, technical reference. `http://www.dealii.org`.

[2] Bangerth, W., Hartmann, R. & Kanschat, G. (2007). `deal.II` – a general purpose object oriented finite element library, *ACM Trans. Math. Softw.*, pp. 24/1-24/27.

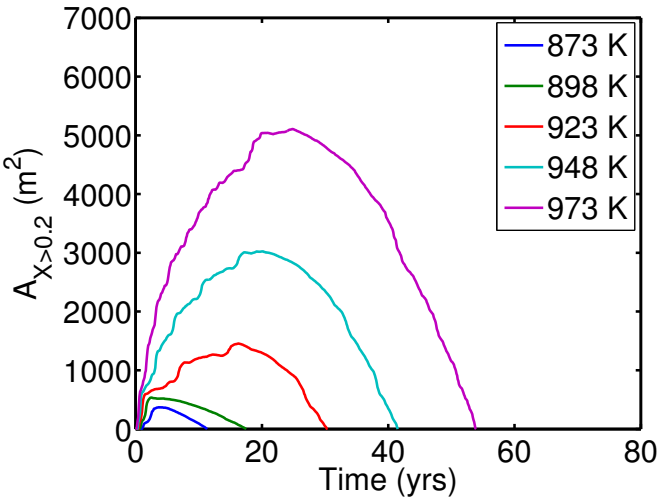


Figure 20: Max. $A_{X>0.2}$ vs. time.

[3] Annen, C., Blundy, J.D. & Sparks, R.S.J. (2006). The genesis of intermediate and silicic magmas in deep crustal hot zones. *Journal of Petrology* **47**(**3**), 505-539.

[4] Dufek, J. & Bergantz, G. W. (2005). Lower crustal genesis and preservation: a stochastic framework fort the evaluation of basalt-crust interaction. Journal of Petrology **46**(**11**), 2167-2195.

[5] Lange, R.A. & Hetland, E. A. (2010). The Thermal Evolution of the Lower Arc Crust During Basalt Emplacement: Importance of Melt Advection out of the Lower Crust in Maintaining a Relatively Cool Steady State Geotherm. AGU Fall Meeting Abstracts. V13G-01.

[6] Lim, S., Hetland, E. A. & Lange, R.A. (2012). Numerical models of transient partial melting of the lower crust during repeated emplacement of basalt sills and subsequent cooling due to advection of melt out of the lower crust. AGU Fall Meeting Abstracts. T13G-2708.

[7] Hu, H. & Argyropoulos, S.A. (1996). Mathematical modeling of solidification and melting: a review. Modeling Simul. Mater. Sci. Eng. **4**, 371-396.

[8] Janssen, B. & Wick, T. (2010). `deal.II` compact course. *University of Heidelberg, Institute of Applied Mathematics*. Session 1-7.

[9] Vigneresse, J. L., Tikoff, B. & Ameglio, L. (1999). Modification of the regional stress field by magma intrusion and formation of tabular granitic plutons. Elsevier. **302**(**3**), pp. 203-224(22).

[10] Kelly, D.W., Gago, J. P. & Zienkiewicz, O.C. (1983). A posteriori error analysis and adaptive processes in the finite element method: Part I Error analysis. Int. J. Num. Meth. Eng. **19**, 1593-1619.

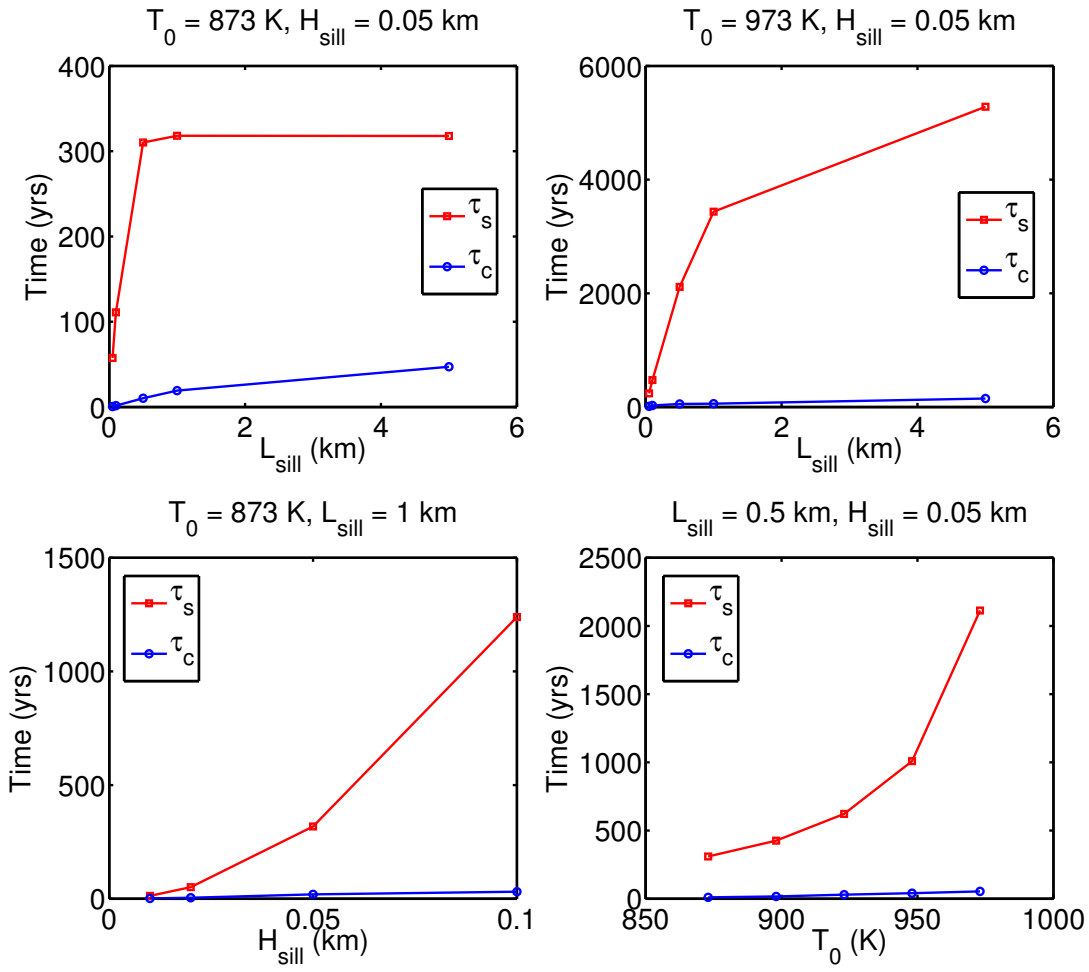


Figure 21: Top left: τ_s and τ_c vs. L_{sill} with $T_0 = 873$ K; Top right: τ_s and τ_c vs. L_{sill} with $T_0 = 973$ K; Bottom left: τ_s and τ_c vs. H_{sill} with $T_0 = 873$ K; Bottom right: τ_s and τ_c vs. T_0.

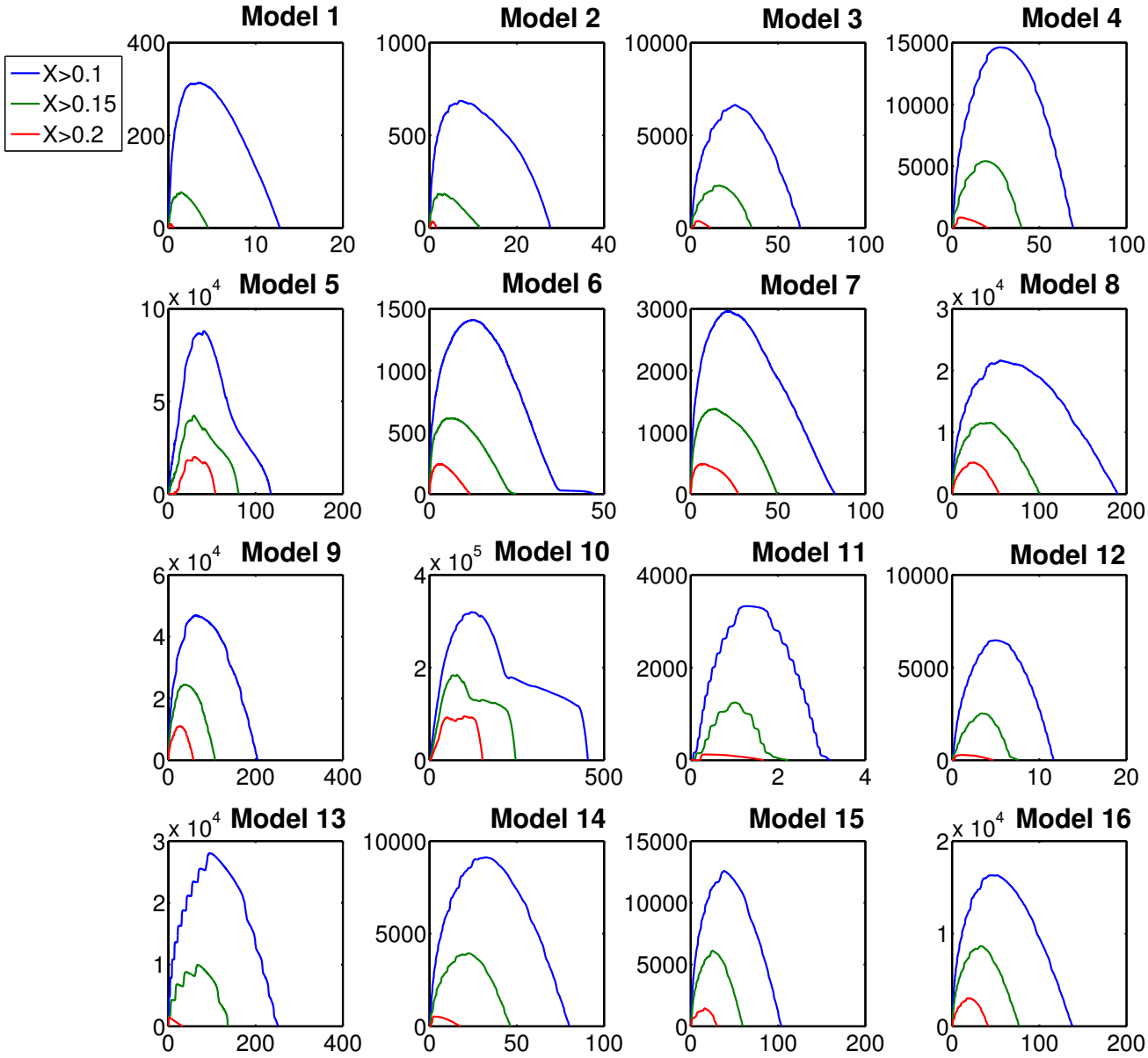
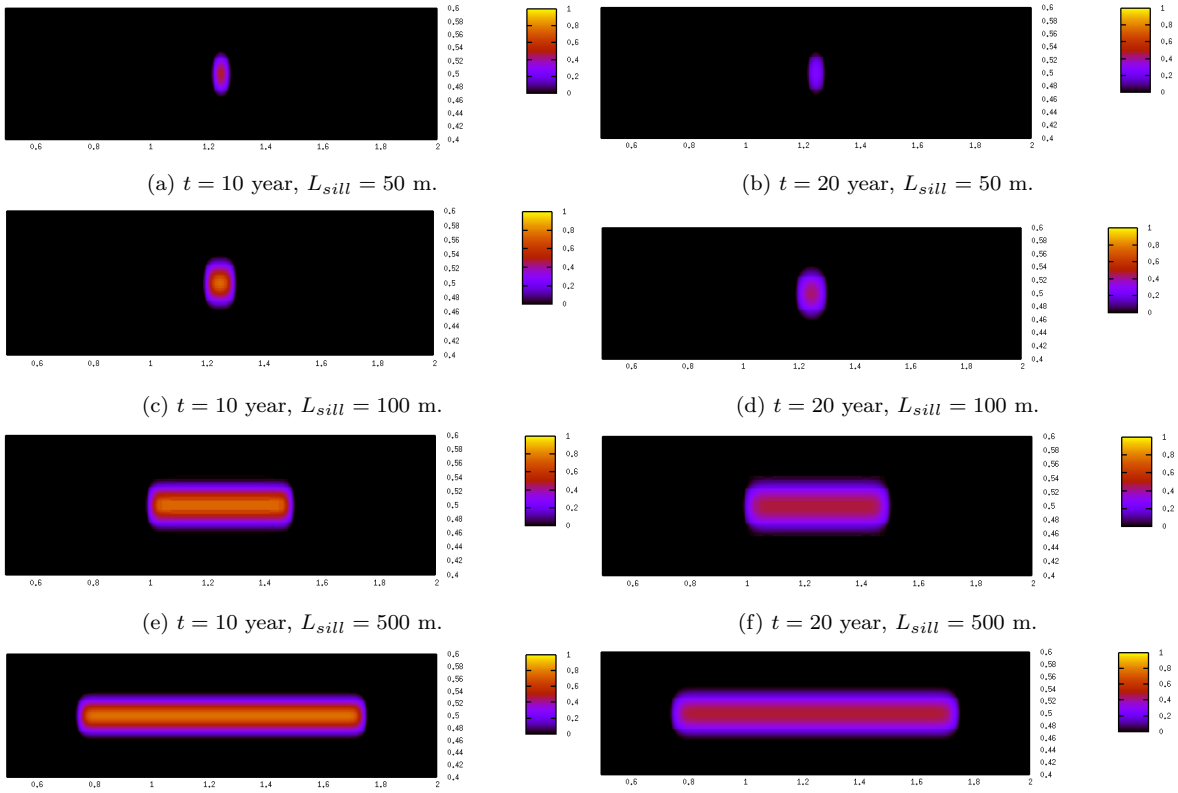


Figure 22: Crustal area (m^2) vs. time (years) plots for Models 1-16.



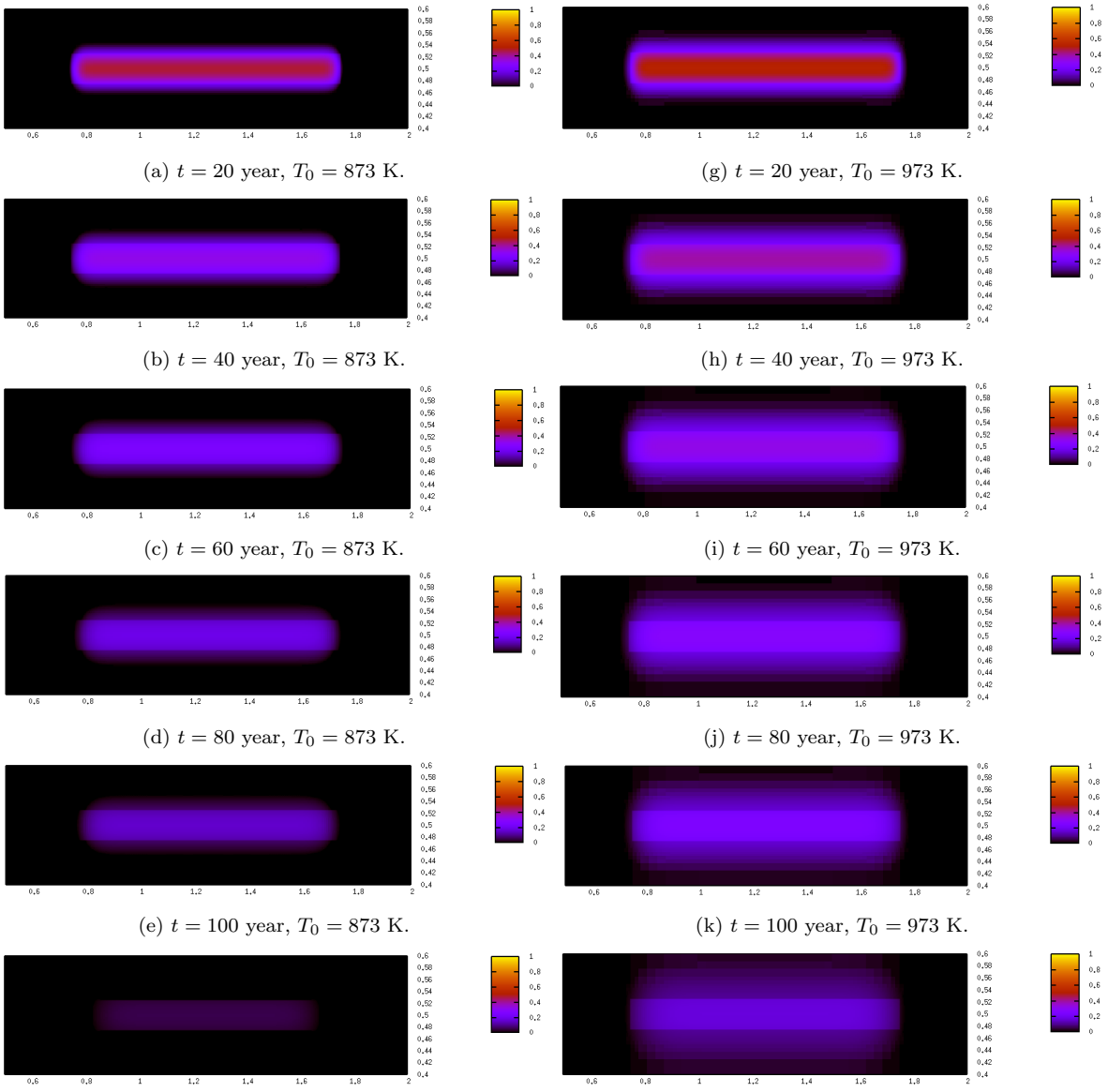
(a) $t = 10$ year, $L_{sill} = 50$ m. (b) $t = 20$ year, $L_{sill} = 50$ m.

(c) $t = 10$ year, $L_{sill} = 100$ m. (d) $t = 20$ year, $L_{sill} = 100$ m.

(e) $t = 10$ year, $L_{sill} = 500$ m. (f) $t = 20$ year, $L_{sill} = 500$ m.

(g) $t = 10$ year, $L_{sill} = 1000$ m. (h) $t = 20$ year, $L_{sill} = 1000$ m.

Figure 23: Melt fraction, X, following emplacement of a sill with $H_{sill} = 50$ m and L_{sill} varied, where $L_{sill} = 50$ m (a-b), 100 m (c-d), 500 m (e-f), and 1000 m (g-h), at times, $t = 10$ and $t = 20$ years in a crust with temperature, $T_0 = 873$ K.



(a) $t = 20$ year, $T_0 = 873$ K.

(g) $t = 20$ year, $T_0 = 973$ K.

(b) $t = 40$ year, $T_0 = 873$ K.

(h) $t = 40$ year, $T_0 = 973$ K.

(c) $t = 60$ year, $T_0 = 873$ K.

(i) $t = 60$ year, $T_0 = 973$ K.

(d) $t = 80$ year, $T_0 = 873$ K.

(j) $t = 80$ year, $T_0 = 973$ K.

(e) $t = 100$ year, $T_0 = 873$ K.

(k) $t = 100$ year, $T_0 = 973$ K.

(f) $t = 200$ year, $T_0 = 873$ K.

(l) $t = 200$ year, $T_0 = 973$ K.

Figure 24: Melt fraction, X, following emplacement at a 1 km by 50 m sill in a crust with temperatures $T_0 = 873$ K (a-f) and $T_0 = 973$ K (g-l) at times, $t = 20,\ 40,\ 60,\ 80,\ 100$, and 200 years.

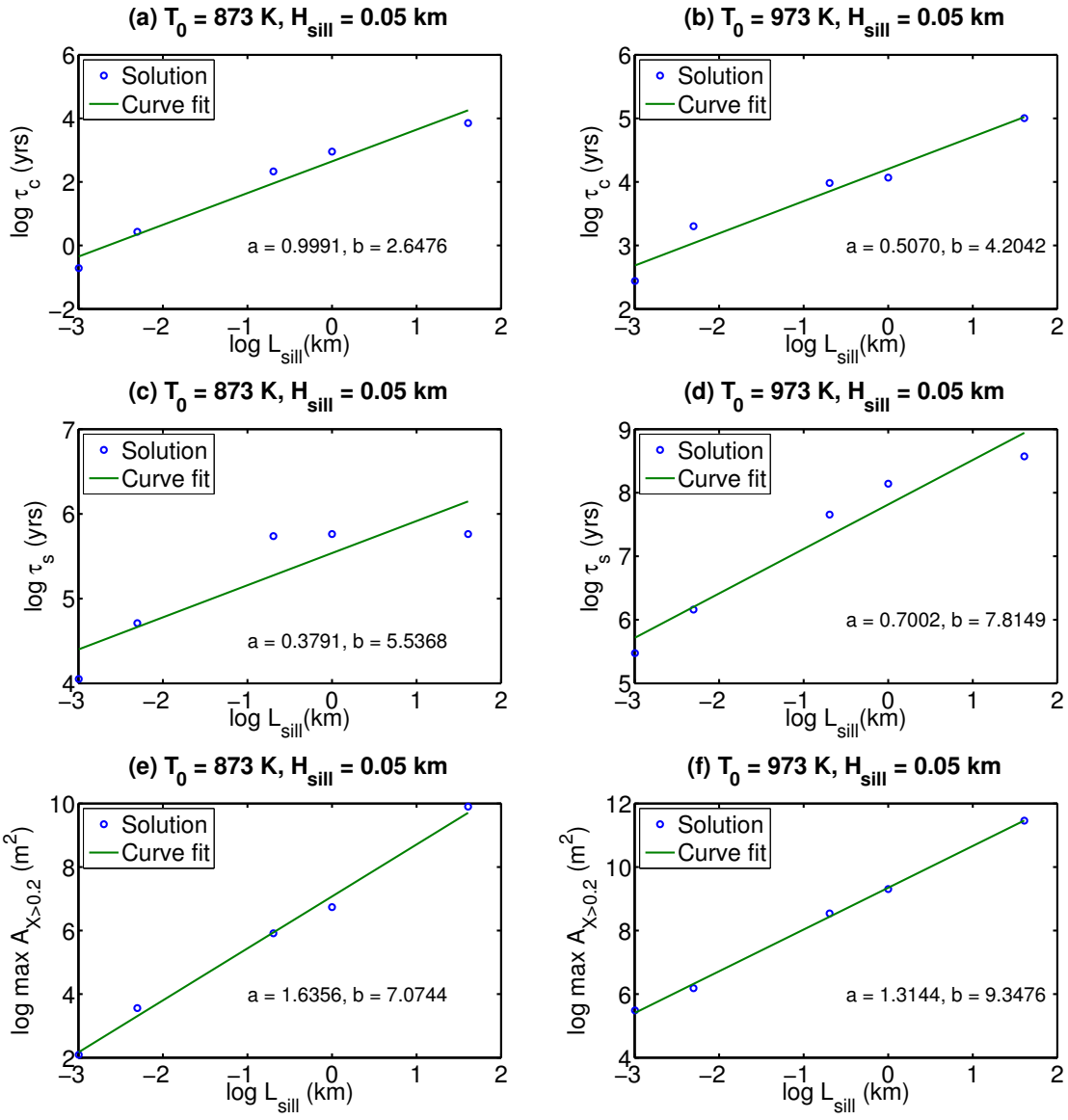


Figure 25: Power laws deduced from simulations for the variables τ_c (a-b), τ_s (c-d), and max. $A_{X>0.2}$ (e-f) vs. L_{sill}. The power laws seem to fit the solution well in (e-f), reasonably well in (a-b), and poorly in (c-d). Here a = slope of the graph and b = y-intercept.

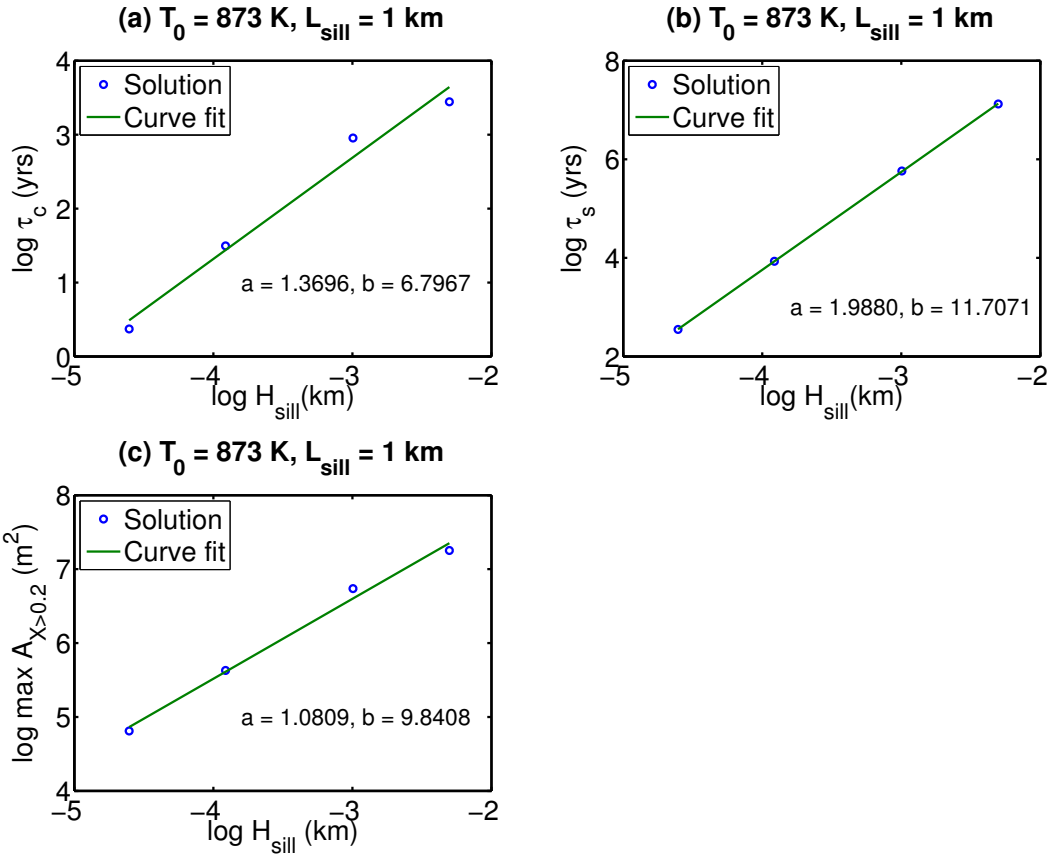


Figure 26: Power laws deduced from simulations for the variables τ_c (a) , τ_s (b), and max. $A_{X>0.2}$ (c) vs. H_{sill}. Here a = slope of the graph and b = y-intercept.

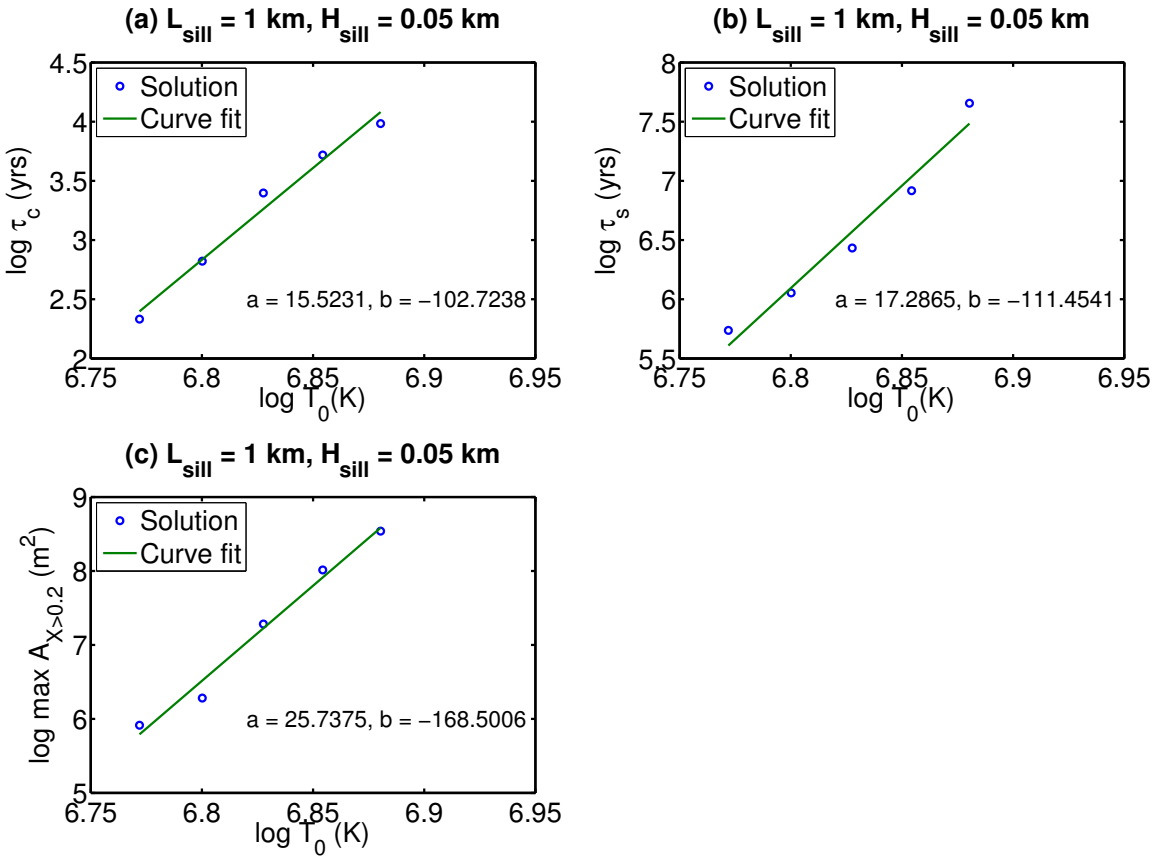


Figure 27: Power laws deduced from simulations for the variables τ_c (a), τ_s (b), and max. $A_{X>0.2}$ (c) vs. T_0. Here a = slope of the graph and b = y-intercept.

[11] Annen, C. & Sparks, R.S.J. (2002). Effects of repetitive emplacement of basaltic intrusions on thermal evolution and melt generation in the crust. *Earth and Planetary Science Letters* **203**, 937-955.

[12] Huppert, H.E. & Sparks, S.J. (1988). The generation of granitic magmas by intrusion of basalt into continental crust. *Journal of Petrology* **29**(**3**), 599-624.

[13] Eriksson, K. , Estep, D., Hansbo, P. & Johnson, C. (1996). Computational differential equations, *Cambridge University Press*.

[14] Bradie, B., (2006) A friendly introduction to numerical analysis, *Pearson Prentice Hall*.

[15] Gratsch, T. & Bathe, K.J. (2004). A posteriori error estimation techniques in practical finite element analysis, *Elsevier*, doi:10.1016/j.compstruc.2004.08.011

[16] Verfurth, R. (2003). A posteriori error estimates for finite element discretizations of the heat equation, *AMS* **40**(**3**), 195-212.

[17] Flaherty, J.E. (2012). Finite element analysis course notes. RPI. CSCI, MATH 6860: Finite element analysis.

[18] Farlow, S.J. (1993). Partial differential equations for scientists and engineers, *Dover Publications*.

Appendices

A Derivation of Analytical Solution to Validation Problems

We consider the problem introduced in Section 5: Find the time-dependent solution $T(x, y, t)$ on $\Omega = [0, L] \times [0, H]$ to the 2D heat balance equation

$$\partial_t T = k^* \nabla^2 T, \tag{24}$$

where $k^* = k/C_p\rho$ is a constant. The solution is subjected to the initial and boundary conditions as stated in (19) - (21).

We find an analytical solution using separation of variables. Since the boundary conditions are not homogeneous, separation of variables cannot be applied directly to find the solution. Instead, we transform the original problem to the one with homogeneous boundary conditions by

$$T(x,y,t) = T_0 - \frac{g_0 H}{2} + g_0 y + w(x,y,t), \tag{25}$$

where

$$w_t = \kappa^*(w_{xx} + w_{yy}), \tag{26}$$

which is subjected to the boundary conditions

$$w(x,0,t) = w(x,H,t) = w_y(0,y,t) = w_y(L,y,t) = 0, \tag{27}$$

and the initial condition

$$w(x,y,0) = \begin{cases} T_{sill} - T_0 + \frac{g_0 H}{2} - g_0 y & \text{in } [L_1, L_2] \times [H_1, H_2], \\ 0 & \text{elsewhere.} \end{cases} \tag{28}$$

We find a solution to the transformed problem of the form

$$w(x,y,t) = X(x)Y(y)T(t) \tag{29}$$

to

$$w_t = k^*(w_{xx} + w_{yy}). \tag{30}$$

Substituting (29) into (30) and rearranging terms, we find

$$X(x)Y(y)T'(t) = k^*\{X''\ Y(y)\ T(t) + X(x)Y''(y)T(t)\}. \tag{31}$$

Therefore,

$$\frac{T'(t)}{k^* T(t)} = \frac{X''(x)}{X(x)} + \frac{Y''(y)}{Y(y)} = -\lambda, \tag{32}$$

and

$$T'(t) = -\lambda\ k^* T(t) \implies T(t) = Ae^{-\lambda k^* t}, \tag{33}$$

$$\frac{X''(x)}{X(x)} = -\lambda - \frac{Y''(y)}{Y(y)} = -\beta, \tag{34}$$

where

$$X''(x) = -\beta X(x) \implies X(x) = B\cos(\sqrt{\beta}x) + C\sin(\sqrt{\beta}x), \tag{35}$$

$$Y''(y) = (\beta - \lambda)Y(y) \implies Y(y) = D\cos(\sqrt{\lambda - \beta}y) + E\sin(\sqrt{\lambda - \beta}y), \tag{36}$$

and

$$Y(0) = 0 \implies D = 0 \implies Y(y) = E\sin(\sqrt{\lambda - \beta}y), \tag{37}$$

$$Y(H) = 0 \implies E\sin(\sqrt{\lambda - \beta}H) = 0 \implies \sqrt{\lambda - \beta}H = m\pi \implies \lambda - \beta = \left(\frac{m\pi}{H}\right)^2,\ m = 1, 2, 3. \tag{38}$$

Plugging in the $X(x)$ and $Y(y)$ obtained and applying the boundary conditions, we get

$$w(x, y, t) = Ae^{-\lambda k^* t}\{B\cos(\sqrt{\beta}x) + C\sin(\sqrt{\beta}x)\}E\sin\left(\frac{m\pi y}{H}\right), \tag{39}$$

$$w_y(0, y, t) = ABE\ e^{-\lambda k^* t}\frac{m\pi}{H}\cos\left(\frac{m\pi y}{H}\right) = 0, \tag{40}$$

$$w_y(L, y, t) = Ae^{-\lambda k^* t}\frac{m\pi}{H}\left[B\cos(\sqrt{\beta}L) + C\sin(\sqrt{\beta}L)\right]\left[E\cos\left(\frac{m\pi y}{H}\right)\right] = 0, \tag{41}$$

$$B = 0 \implies C\sin(\sqrt{\beta}L) = 0 \implies \beta = \left(\frac{n\pi}{L}\right),\ n = 1, 2, 3, ..., \tag{42}$$

$$\lambda - \left(\frac{n\pi}{L}\right)^2 = \left(\frac{m\pi}{H}\right)^2 \implies \lambda_{mn} = \left(\frac{n\pi}{L}\right)^2 + \left(\frac{m\pi}{H}\right)^2,\ n, m = 1, 2, 3, ..., \tag{43}$$

$$w_{nm}(x, y, t) = A_{nm}C_{nm}E_{nm}\ e^{-\left[\left(\frac{n\pi}{L}\right)^2 + \left(\frac{m\pi}{H}\right)^2\right]k^* t}\sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{H}\right), \tag{44}$$

$$w(x,y,t) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} \tilde{C}_{nm}\; e^{-\left[\left(\frac{n\pi}{L}\right)^2+\left(\frac{m\pi}{H}\right)^2\right]k^* t} \sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{H}\right). \tag{45}$$

Applying the initial condition, we find $\tilde{C}_{nm}$.

$$w(x,y,0) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} \tilde{C}_{nm} \sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{H}\right) = \begin{cases} T_{sill} - T_0 + \frac{g_0 H}{2} - g_0 y & \text{on } \Omega_{sill} = [L_1, L_2] \text{ x } [H_1, H_2], \\ 0 & \text{elsewhere,} \end{cases} \tag{46}$$

where

$$\tilde{C}_{nm} = \frac{4(T_{sill} - T_0 + \frac{g_0 H}{2} - g_0 y)}{nm\pi^2}\left[\cos\left(\frac{n\pi L_2}{L}\right) - \cos\left(\frac{n\pi L_1}{L}\right)\right]\left[\cos\left(\frac{m\pi H_2}{H}\right) - \cos\left(\frac{m\pi H_1}{H}\right)\right]. \tag{47}$$

Hence, the desired solution is

$$w(x,y,t) = \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} \tilde{C}_{nm}\; e^{-\left[\left(\frac{n\pi}{L}\right)^2+\left(\frac{m\pi}{H}\right)^2\right]k^* t} \sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi y}{H}\right), \tag{48}$$

where

$$\tilde{C}_{nm} = \frac{4(T_{sill} - T_0 + \frac{g_0 H}{2} - g_0 y)}{nm\pi^2}\left[\cos\left(\frac{n\pi L_2}{L}\right) - \cos\left(\frac{n\pi L_1}{L}\right)\right]\left[\cos\left(\frac{m\pi H_2}{H}\right) - \cos\left(\frac{m\pi H_1}{H}\right)\right]. \tag{49}$$

Finally, the complete solution to the validation problems in Section 5 is

$$T(x,y,t) = T_0 - \frac{g_0 H}{2} + g_0 y + w(x,y,t). \tag{50}$$

B Source Codes

B.1 Verification codes for validation models

```
1  ===================================== compute.cc ======================================
2  #include <fstream>
3  #include <iostream>
4  #include <cmath>
5
6  // REQUIRES:  t > 0, x and y must be in the range of the domain.
```

```
7  // MODIFIES: none.
8  // EFFECTS: compute the analytical solution up to N terms for a specified geotherm g_0.
9  double compute (double t, double x, double y){
10   double pi = 3.14159265;
11   double series_soln = 0;
12   const double sec = (365.25*24*3600);
13   const double k = 8.154421735e19;
14   const double rho = 3.05e12;
15   const double cp = 1390*sec*sec/1000000;
16   const double k1 = k/(rho*cp);
17   const double T_o = 1558.;
18   const double L1 = 1.0;
19   const double L2 = 1.5;
20   const double W1 = 0.475;
21   const double W2 = 0.525;
22   const double L = 2.5;
23   const double H = 1.0;
24   const double back_temp = 873;
25   const double grad = 0; /* specify which validation models to compute */
26   const int N = 50;  /* specify number of terms computed */
27
28   for(int n=1; n<N; n++){
29            for(int m=1; m<N; m++){
30                 double bnm = (4*(T_o - back_temp + grad*H/2 - grad*y)/(n*m*pi*pi))*(cos(W2*m*pi/H) -
31                              cos(W1*m*pi/H))*(cos(L2*n*pi/L)-cos(L1*n*pi/L));
32                 series_soln += bnm*exp(-(((n*n/(L*L)+m*m/(H*H))*pi*pi)*k1*t))
33                                   * sin(n*pi*x/L)*sin(m*pi*y/H);
34            }
35   }
36   series_soln += back_temp - grad*H/2 + grad*y;
37
38   return series_soln;
39 }
40
41 int main ()
42 {
43    int k = 0;
44    const double dt = 0.1;
45    std::ofstream file;
46    file.open("series_solution.txt");
47    while (k <= 500){   /* run for 500 time steps */
48       std::cout << "Time: " << k*0.1 << std::endl;
49
50       /* compute series solution at 4 different points in the domain */
51       file << k
52            << " " << compute(k*dt, 1.25, 0.525)
53            << " " << compute(k*dt, 1.25, 0.530)
54            << " " << compute(k*dt, 1.5, 0.525)
55            << " " << compute(k*dt, 1.5, 0.530)
```

```
            << std::endl;
       k+=1;
    }
    file.close();
    return 0;
}
```

B.2 Program codes

Note: `Makefile` is available upon installation of `deal.II`.

```
================================== parameter.prm ================================
  # List of parameters for numerical experiments

  # Specify the test case we want to run:
  # TEST_ZERO - test case with analytical solution
  # TEST_ONE - test case with variable k and a melt profile
  set TestCase = TEST_ZERO

  subsection Physical data
    # Declare the initial time, final time, time step size, and domain length
    set time = 0.
    set final_time   = 50.
    set time_step     = 0.1

    # Declare the onset temperature of solid and liquid in the crust (TC) and basalt (TM)
    # Irrelevant if Test0 is set
    set TC_solid = 1025.0
    set TC_liquid = 1858.0
    set TM_solid = 999.0
    set TM_interm = 1366.0
    set TM_liquid = 1513.0
  end

  subsection Space discretization
    # Declare the data that is relevant to the space discretization
    set deg  = 2
    set freq = 50
    set ini_refinement_level = 10
    set n_pre_refinement_steps = 7

    # Specify the refinement criterion to use for adaptive remeshing:
    # 1- global refinement, 2- KellyErrorEstimator, 3- 2ndDerivativeApproximation
    set mode = 2

    # If mode = 1, specify the levels of global refinement
    set level = 3
  end

```

```
100
101 ==================================== source.cc ====================================
102 /* Disclaimer: This program is developed by modifying the existing tutorial programs
103                in deal.II documentation. A large parts of the codes in this program
104                are inherited from the tutorial programs for the purpose of solving
105                a problem in an application                */
106
107 /* Copyright (C) 2009-2012 by deal.II authors              */
108
109 /* Instruction: To perform numerical experiments, either change the variables in
110    parameter.prm or in the namespace SillData @line 78     */
111
112 #include <deal.II/base/parameter_handler.h>
113 #include <deal.II/base/data_out_base.h>
114 #include <base/quadrature_lib.h>
115 #include <base/function.h>
116 #include <base/logstream.h>
117 #include <base/utilities.h>
118 #include <base/smartpointer.h>
119 #include <base/timer.h>
120 #include <lac/vector.h>
121 #include <lac/full_matrix.h>
122 #include <lac/sparse_matrix.h>
123 #include <lac/solver_cg.h>
124 #include <lac/compressed_sparsity_pattern.h>
125 #include <lac/precondition.h>
126 #include <lac/filtered_matrix.h>
127 #include <lac/constraint_matrix.h>
128 #include <grid/tria.h>
129 #include <grid/grid_generator.h>
130 #include <grid/grid_tools.h>
131 #include <grid/tria_accessor.h>
132 #include <grid/tria_iterator.h>
133 #include <grid/grid_refinement.h>
134 #include <deal.II/grid/grid_out.h>
135 #include <deal.II/dofs/dof_tools.h>
136 #include <dofs/dof_handler.h>
137 #include <dofs/dof_accessor.h>
138 #include <dofs/dof_tools.h>
139 #include <dofs/dof_constraints.h>
140 #include <dofs/dof_renumbering.h>
141 #include <fe/fe_q.h>
142 #include <fe/fe_values.h>
143 #include <fe/mapping_q.h>
144 #include <numerics/vectors.h>
145 #include <numerics/matrices.h>
146 #include <numerics/data_out.h>
147 #include <numerics/error_estimator.h>
```

```
148 #include <numerics/solution_transfer.h>
149 #include <numerics/derivative_approximation.h>
150
151 #include <cmath>
152 #include <fstream>
153 #include <iostream>
154 #include <list>
155 #include <iomanip>
156 #include <algorithm>
157
158 namespace Earth
159 {
160  using namespace dealii;
161
162  namespace EquationConstants
163  {                                          /* UNITS             */
164     const double sec = (365.25*24*3600);    /* s                 */
165     const double k = 8.154421735e19;        /* kg*km/(K*yr^3)    */
166     const double rho_cs = 3.05e12;          /* kg/km^3           */
167     const double rho_cl = 2.3e12;           /* kg/km^3           */
168     const double rho_bs = 3.1e12;           /* kg/km^3           */
169     const double rho_bl = 2.83e12;          /* kg/km^3           */
170     const double L_b = (4.0e5)*sec*sec/1000000; /* km^2/(K*yr^2)     */
171     const double L_c = (3.5e5)*sec*sec/1000000; /* km^2/(K*yr^2)     */
172     const double cp_b = 1480*sec*sec/1000000;   /* km^2/yr^2         */
173     const double cp_c = 1390*sec*sec/1000000;   /* km^2/yr^2         */
174     double kstar = 0;
175  }
176
177  namespace SillData
178  {
179    const double DomainLength = 2.5;          /* km                */
180    const double DomainHeight = 1.0;          /* km                */
181    const double L1 = 1.0;                    /* km                */
182    const double L2 = 1.5;                    /* km                */
183    const double W1 = 0.475;                  /* km                */
184    const double W2 = 0.525;                  /* km                */
185    const double ini_temp = 1558.;            /* K                 */
186    const double geotherm = 0.;              /* K/km              */
187    const double back_temp = 873.;            /* K                 */
188  }
189
190 namespace Parameters
191  {
192    enum TestCase
193    {
194       TEST_ZERO,
195       TEST_ONE
196    };
```

```
197
198     class DataInput
199     {
200       public:
201          DataInput ();
202          ~DataInput ();
203          void read_data (const char *filename);
204          TestCase test;
205          double time,
206          final_time,
207          time_step,
208          TC_liquid,
209          TC_solid,
210          TM_liquid,
211          TM_interm,
212          TM_solid,
213          dm_meltfrac;
214          unsigned int deg;
215          unsigned int freq,
216          mode,
217          ini_refinement_level,
218          n_pre_refinement_steps,
219          level;
220       protected:
221          ParameterHandler prm;
222     };
223
224     DataInput::DataInput()
225     {
226          prm.declare_entry ("TestCase", "TEST_ONE",
227                             Patterns::Selection ("TEST_ZERO|TEST_ONE"),
228                             "Used to select the test case that we are going "
229                             "to use. ");
230
231          prm.enter_subsection ("Physical data");
232          {
233             prm.declare_entry ("time", "0.", Patterns::Double (0.),
234                                "Time of simulation. ");
235             prm.declare_entry ("final_time", "0.", Patterns::Double (0.),
236                                "Final time of simulation. ");
237             prm.declare_entry ("time_step", "0.", Patterns::Double (0.),
238                                "Time step of simulation. ");
239             prm.declare_entry ("TC_solid", "0.", Patterns::Double (0.),
240                                "Solid temp for crust. ");
241             prm.declare_entry ("TC_liquid", "0.", Patterns::Double (0.),
242                                "Liquid temp for crust. ");
243             prm.declare_entry ("TM_solid", "0.", Patterns::Double (0.),
244                                "Solid temp for sill. ");
245             prm.declare_entry ("TM_interm", "0.", Patterns::Double (0.),
```

```
246                               "Intermediate temp for sill. ");
247            prm.declare_entry ("TM_liquid", "0.", Patterns::Double (0.),
248                               "Liquid temp for sill. ");
249
250         }
251         prm.leave_subsection();
252
253         prm.enter_subsection ("Space discretization");
254         {
255            prm.declare_entry ("deg", "1", Patterns::Integer (1,3),
256                               "Polynomial degree for the temperature space");
257            prm.declare_entry ("freq", "0", Patterns::Integer (0,10000),
258                               "Frequency of pre-time adaptive refinement");
259            prm.declare_entry ("ini_refinement_level", "2",
260                               Patterns::Integer (2,1000),
261                               "Initial number of cells = this*this");
262            prm.declare_entry ("n_pre_refinement_steps", "0",
263                               Patterns::Integer (0,10),
264                               "Number of pre adaptive refinement steps");
265            prm.declare_entry ("mode", "1", Patterns::Integer (1, 5),
266                               "Choice of refinement strategy. ");
267            prm.declare_entry ("level", "0", Patterns::Integer (0, 10),
268                               "Number of global refinements for mode 1. ");
269         }
270         prm.leave_subsection();
271
272      }
273
274      DataInput::~DataInput()
275      {}
276
277      void DataInput::read_data (const char *filename)
278      {
279           std::ifstream file (filename);
280           AssertThrow (file, ExcFileNotOpen (filename));
281
282           prm.read_input (file);
283
284           if (prm.get ("TestCase") == std::string ("TEST_ZERO")){
285              test = TEST_ZERO;
286           }
287           else{
288              test = TEST_ONE;
289           }
290
291           prm.enter_subsection ("Physical data");
292           {
293              time = prm.get_double ("time");
294              final_time = prm.get_double ("final_time");
```

```
295             time_step = prm.get_double ("time_step");
296             TC_solid = prm.get_double ("TC_solid");
297             TC_liquid = prm.get_double ("TC_liquid");
298             TM_solid = prm.get_double ("TM_solid");
299             TM_interm = prm.get_double ("TM_interm");
300             TM_liquid = prm.get_double ("TM_liquid");
301          }
302          prm.leave_subsection();
303
304          prm.enter_subsection ("Space discretization");
305          {
306             deg = prm.get_integer ("deg");
307             freq = prm.get_integer ("freq");
308             ini_refinement_level = prm.get_integer ("ini_refinement_level");
309             n_pre_refinement_steps = prm.get_integer ("n_pre_refinement_steps");
310             mode = prm.get_integer ("mode");
311             level = prm.get_integer ("level");
312          }
313          prm.leave_subsection();
314
315       }
316    }
317
318
319    /* class denoting the initial conditions for case with zero geotherm */
320    class InitialValues_ZeroGeotherm : public Function<2>
321    {
322      public:
323          InitialValues_ZeroGeotherm (const unsigned int n_components = 2,
324                  const double time = 0.)
325                  :
326                  Function<2>(n_components, time)
327          {}
328          virtual double value (const Point<2> &p,
329                  const unsigned int component = 0) const;
330          virtual void vector_value (const Point<2> &p,
331                  Vector<double> &value) const;
332    };
333
334    double InitialValues_ZeroGeotherm::value (const Point<2> &p,
335          const unsigned int /*component*/) const
336    {
337          const double x = p[0];
338          const double y = p[1];
339
340          if(x>=SillData::L1 && x<=SillData::L2 && y>= SillData::W1
341                  && y<=SillData::W2){
342                  return SillData::ini_temp;
343          }
```

```
344         else{
345                 return SillData::back_temp;
346         }
347 }
348
349 void InitialValues_ZeroGeotherm::vector_value (const Point<2> &p,
350         Vector<double> &value) const
351 {
352         for (unsigned int c=0; c<this->n_components; ++c){
353                 value(c) = InitialValues_ZeroGeotherm::value(p,c);
354         }
355 }
356
357 /* class denoting the initial conditions for case with nonzero geotherm */
358 class InitialValues_Geotherm : public Function<2>
359 {
360   public:
361         InitialValues_Geotherm (const unsigned int n_components = 2,
362                 const double time = 0.)
363                 :
364                 Function<2>(n_components, time)
365         {}
366         virtual double value (const Point<2> &p,
367                 const unsigned int component = 0) const;
368         virtual void vector_value (const Point<2> &p,
369                 Vector<double> &value) const;
370 };
371
372 double InitialValues_Geotherm::value (const Point<2> &p,
373         const unsigned int /*component*/) const
374 {
375         const double x = p[0];
376         const double y = p[1];
377
378         if(x>=SillData::L1 && x<=SillData::L2 && y>= SillData::W1
379                 && y<=SillData::W2){
380                 return SillData::ini_temp;
381         }
382         else{
383                 return (SillData::back_temp-SillData::geotherm*SillData::DomainHeight/2) +
384                         SillData::geotherm*y;
385         }
386 }
387
388 void InitialValues_Geotherm::vector_value (const Point<2> &p,
389         Vector<double> &value) const
390 {
391         for (unsigned int c=0; c<this->n_components; ++c){
392                 value(c) = InitialValues_Geotherm::value(p,c);
```

```
393         }
394   }
395
396
397   /* class denoting the boundary conditions for case with zero geotherm */
398   class BoundaryData_ZeroGeotherm : public Function<2>
399   {
400      public:
401         BoundaryData_ZeroGeotherm (const double time = 0.)  : Function<2>(1, time) {}
402         virtual double value (const Point<2>   &p,
403                               const unsigned int  component = 0) const;
404         virtual void value_list (const std::vector<Point<2> > &points,
405                                  std::vector<double>          &values,
406                                  const unsigned int component = 0) const;
407   };
408
409   double BoundaryData_ZeroGeotherm::value (const Point<2>   & /*p*/,
410         const unsigned int /*component*/ ) const
411   {
412         return SillData::back_temp;
413   }
414
415   void BoundaryData_ZeroGeotherm::value_list(const std::vector<Point<2> > &points,
416                                 std::vector<double>          &values,
417                                 const unsigned int component) const
418   {
419          Assert (values.size() == points.size(),
420                 ExcDimensionMismatch (values.size(), points.size()));
421          Assert (component == 0,
422                 ExcIndexRange (component, 0, 1));
423
424         const unsigned int n_points = points.size();
425
426         for (unsigned int i=0; i<n_points; ++i)
427         {
428                 values[i] = BoundaryData_ZeroGeotherm::value(points[i]);
429         }
430   }
431
432   /* class denoting the boundary conditions for case with nonzero geotherm */
433   class BoundaryData_Geotherm : public Function<2>
434   {
435      public:
436         BoundaryData_Geotherm (const double time = 0.)  : Function<2>(1, time) {}
437         virtual double value (const Point<2>   &p,
438                               const unsigned int  component = 0) const;
439         virtual void value_list (const std::vector<Point<2> > &points,
440                                  std::vector<double>          &values,
441                                  const unsigned int component = 0) const;
```

```
442 };
443
444 double BoundaryData_Geotherm::value (const Point<2>   &p,
445         const unsigned int /*component*/ ) const
446 {
447         const double x = p[0];
448         const double y = p[1];
449
450         if(x>=SillData::L1 && x<=SillData::L2 && y>= SillData::W1
451                 && y<=SillData::W2){
452                 return SillData::ini_temp;
453         }
454         else{
455                 return (SillData::back_temp-SillData::geotherm*SillData::DomainHeight/2) +
456                         SillData::geotherm*y;
457         }
458 }
459
460 void BoundaryData_Geotherm::value_list(const std::vector<Point<2> > &points,
461                                 std::vector<double>          &values,
462                                 const unsigned int component) const
463 {
464          Assert (values.size() == points.size(),
465                 ExcDimensionMismatch (values.size(), points.size()));
466          Assert (component == 0,
467                 ExcIndexRange (component, 0, 1));
468
469         const unsigned int n_points = points.size();
470
471         for (unsigned int i=0; i<n_points; ++i)
472         {
473                 values[i] = BoundaryData_Geotherm::value(points[i]);
474         }
475 }
476
477 /* class denoting the right hand side */
478 class RightHandSide : public Function<2>
479 {
480   public:
481         RightHandSide (double time = 0.)  : Function<2>(1, time) {}
482         virtual double value (const Point<2>   &p,
483                               const unsigned int  component = 0) const;
484         virtual void value_list (const std::vector<Point<2> > &points,
485                                  std::vector<double>          &values,
486                                  const unsigned int component = 0) const;
487 };
488
489 double RightHandSide::value (const Point<2>   &/*p*/,
490                              const unsigned int /*component*/ ) const
```

```
491 {
492         return 0;
493 }
494
495 void RightHandSide::value_list(const std::vector<Point<2> > &points,
496                                std::vector<double>          &values,
497                                const unsigned int component) const
498 {
499         Assert (values.size() == points.size(),
500         ExcDimensionMismatch (values.size(), points.size()));
501         Assert (component == 0,
502         ExcIndexRange (component, 0, 1));
503
504         const unsigned int n_points = points.size();
505
506         for (unsigned int i=0; i<n_points; ++i){
507                 values[i] = RightHandSide::value(points[i]);
508         }
509 }
510
511 /* the main class */
512 class Model
513 {
514   public:
515     Model (const Parameters::DataInput &data);
516     void run ();
517
518   protected:
519     Parameters::TestCase test;
520     unsigned int deg;
521     unsigned int freq, ini_refinement_level;
522     unsigned int n_pre_refinement_steps, mode, level;
523     double time, time_step;
524     const double final_time;
525     const double TC_0, TC_1, TM_0, TM_1, TM_i;
526
527   private:
528     void make_initial_grid ();
529     void setup_dof ();
530     void refine_by_Kelly ();
531     void refine_by_derivative ();
532     void refine_grid (const unsigned int max_grid_level);
533     void assemble_term ();
534     void compute_oldterm ();
535     void compute_newterm ();
536     void compute_matrix ();
537     void assemble_system ();
538     void solve ();
539     void output_results (const unsigned int timestep_number) const;
```

```
540       double evaluate_soln (double x, double y) const ;
541       double compute_kstar (double temp, unsigned char id);
542
543       Triangulation<2>            triangulation;
544       FE_Q<2>                     fe;
545       DoFHandler<2>               dof_handler;
546
547       SparsityPattern             sparsity_pattern;
548       SparseMatrix<double>        system_matrix;
549
550       ConstraintMatrix            constraints;
551
552       PreconditionSSOR<>          preconditioner;
553
554       const MappingQ<2>           mapping;
555
556       bool                        rebuild_matrix;
557
558       Vector<double>              solution, old_solution, melt_fraction;
559       Vector<double>              system_rhs;
560
561       std::vector<double>         error;
562       double                      theta;
563
564       RightHandSide               rhs;
565       BoundaryData_ZeroGeotherm   boundarydata1;
566       BoundaryData_Geotherm       boundarydata2;
567       InitialValues_ZeroGeotherm  initialvalues1;
568       InitialValues_Geotherm      initialvalues2;
569       TimerOutput                 computing_timer;
570       std::ofstream               summary;
571   };
572
573   Model::Model (const Parameters::DataInput &data)
574      :
575      test (data.test),
576      freq (data.freq),
577      ini_refinement_level (data.ini_refinement_level),
578      n_pre_refinement_steps (data.n_pre_refinement_steps),
579      mode (data.mode),
580      level (data.level),
581      time (data.time),
582      time_step (data.time_step),
583      final_time (data.final_time),
584      TC_0 (data.TC_solid),
585      TC_1 (data.TC_liquid),
586      TM_0 (data.TM_solid),
587      TM_1 (data.TM_liquid),
588      TM_i (data.TM_interm),
```

```
589     fe  (data.deg),
590     dof_handler (triangulation),
591     mapping (4),
592     rebuild_matrix (false),
593     theta (1.0), /* 0 <= theta <= 1
594                     theta = 0 corresp to explicit Euler scheme - 1st order acc
595                     theta = 1 corresp to implicit Euler scheme - 1st order acc
596                     theta = 0.5 corresp to Crank-Nicolson scheme - 2nd order acc
597                   */
598     rhs (time),
599     boundarydata1 (time),
600     boundarydata2 (time),
601     computing_timer (summary, TimerOutput::summary, TimerOutput::wall_times)
602   {}
603
604   /* compute k* at a particular time step using previous solution */
605   double Model::compute_kstar(double temp, unsigned char id)
606   {
607        const double k = EquationConstants::k;
608        const double cp_c = EquationConstants::cp_c;
609        const double cp_b = EquationConstants::cp_b;
610        const double L_c = EquationConstants::L_c;
611        const double L_b = EquationConstants::L_b;
612        const double rho_cs = EquationConstants::rho_cs;
613        const double rho_cl = EquationConstants::rho_cl;
614        const double rho_bs = EquationConstants::rho_bs;
615        const double rho_bl = EquationConstants::rho_bl;
616        double kstar = 10000000;
617        double denom = 1.;
618
619        if(test == Parameters::TEST_ZERO){
620                  kstar = k/(rho_cs*cp_c);
621        }
622        else if(id == 'c'){
623           if(temp >= TC_0 && temp <= TC_1){
624                  double dc_mf = 1/(TC_1 - TC_0);
625                  double meltf = dc_mf*temp-1.230492197;
626                  double rho = meltf*(rho_cl - rho_cs) + rho_cs;
627                  kstar = (k/(rho*cp_c + rho*L_c*dc_mf))/denom;
628           }
629           else if(temp > TC_1){
630                  kstar = (k/(rho_cl*cp_c + rho_cl*L_c*1))/denom;
631           }
632           else{
633                  kstar = (k/(rho_cs*cp_c + rho_cs*L_c*0))/denom;
634           }
635        }
636        else if(id == 'm'){
637           if(temp >= TM_0 && temp <= TM_i){
```

```
638                 double dm_mf = 1.4e-3;
639                 double meltf = dm_mf*temp - 1.3986;
640                 double rho = meltf*(rho_bl - rho_bs) + rho_bs;
641                 kstar = (k/(rho*cp_b + rho*L_b*dm_mf))/denom;
642           }
643           else if(temp >= TM_i && temp <= TM_1){
644                 double dm_mf = 3.307482993e-3;
645                 double meltf = dm_mf*temp - 4.004221768;
646                 double rho = meltf*(rho_bl - rho_bs) + rho_bs;
647                 kstar = (k/(rho*cp_b + rho*L_b*dm_mf))/denom;
648           }
649           else if(temp > TM_1){
650                 kstar = (k/(rho_bl*cp_b + rho_bl*L_b*1))/denom;
651           }
652           else{
653                 kstar = (k/(rho_bs*cp_b + rho_bs*L_b*0))/denom;
654           }
655       }
656       else{
657                 std::cout << " Error in allocating subdomains! " << std::endl;
658       }
659
660       return kstar;
661   }
662
663   /* adaptive refinement with KellyErrorEstimator */
664   void Model::refine_by_Kelly ()
665   {
666       std::cout << "======= Refined adaptively using KellyErrorEstimator ======="
667                   <<  std::endl;
668       Vector<float> estimated_error_per_cell (triangulation.n_active_cells());
669
670       KellyErrorEstimator<2>::estimate (mapping, dof_handler,
671                                         QGauss<1>(3),
672                                         FunctionMap<2>::type(),
673                                         solution,
674                                         estimated_error_per_cell);
675
676       if(time == 0){
677           GridRefinement::refine_and_coarsen_fixed_number (triangulation,
678                                                            estimated_error_per_cell,
679                                                            0.3, 0.1);
680       }
681       else if (time < 1000*time_step){
682           GridRefinement::refine_and_coarsen_fixed_number (triangulation,
683                                                            estimated_error_per_cell,
684                                                            0.005, 0.0025);
685       }
686       else{
```

```
687           GridRefinement::refine_and_coarsen_fixed_number (triangulation,
688                                                            estimated_error_per_cell,
689                                                            0.003, 0.0025);
690       }
691   }
692
693
694   /* adaptive refinement with DerivativeApproximation */
695   void Model::refine_by_derivative ()
696   {
697       std::cout << "======= Refined adaptively using DerivativeApproximation ======="
698                 <<  std::endl;
699
700       Vector<float> gradient_indicator (triangulation.n_active_cells());
701
702       DerivativeApproximation::approximate_second_derivative (mapping,
703                                                               dof_handler,
704                                                               solution,
705                                                               gradient_indicator);
706
707       DoFHandler<2>::active_cell_iterator
708       cell = dof_handler.begin_active(),
709       endc = dof_handler.end();
710       for (unsigned int cell_no=0; cell!=endc; ++cell, ++cell_no)
711           gradient_indicator(cell_no)*=std::pow(cell->diameter(), 3.);
712
713           if(time == 0){
714                 GridRefinement::refine_and_coarsen_fixed_number (triangulation,
715                                                                  gradient_indicator,
716                                                                  0.3, 0.1);
717           }
718           else if (time < 1000*time_step){
719                 GridRefinement::refine_and_coarsen_fixed_number (triangulation,
720                                                                  gradient_indicator,
721                                                                  0.005, 0.0025);
722           }
723           else{
724                 GridRefinement::refine_and_coarsen_fixed_number (triangulation,
725                                                                  gradient_indicator,
726                                                                  0.002, 0.0025);
727          }
728   }
729
730   /* main function for adaptive mesh refinement*/
731   void Model::refine_grid (const unsigned int max_grid_level)
732   {
733      computing_timer.enter_section ("Remeshing");
734
735      if(mode == 1){
```

```
736           std::cout << "======= Refined globally " << level << " times  =======" 
737                     <<  std::endl;
738       }
739       else if(mode == 2){
740           refine_by_Kelly();
741       }
742       else{
743           refine_by_derivative();
744       }
745 
746       if(triangulation.n_levels() > max_grid_level)
747           for(Triangulation<2>::active_cell_iterator
748                   cell = triangulation.begin_active(max_grid_level);
749                   cell != triangulation.end(); ++cell){
750                           cell->clear_refine_flag();
751           }
752 
753           SolutionTransfer<2> soltrans (dof_handler);
754 
755 
756           /* Transfer old soln to new mesh */
757           triangulation.prepare_coarsening_and_refinement();
758 
759           soltrans.prepare_for_coarsening_and_refinement (solution);
760           triangulation.execute_coarsening_and_refinement ();
761 
762           dof_handler.distribute_dofs(fe);
763 
764           constraints.clear();
765           DoFTools::make_hanging_node_constraints (dof_handler, constraints);
766           constraints.close();
767 
768           system_matrix.clear();
769 
770           CompressedSparsityPattern c_sparsity (dof_handler.n_dofs());
771           DoFTools::make_sparsity_pattern (dof_handler, c_sparsity);
772           constraints.condense (c_sparsity);
773           sparsity_pattern.copy_from(c_sparsity);
774 
775 
776           Vector<double> interpolated_soln(dof_handler.n_dofs());
777           soltrans.interpolate(solution, interpolated_soln);
778           solution = interpolated_soln;
779 
780           system_matrix.reinit(sparsity_pattern);
781           old_solution.reinit(dof_handler.n_dofs());
782           system_rhs.reinit(dof_handler.n_dofs());
783 
784           rebuild_matrix = true;
```

```
785
786         computing_timer.exit_section();
787     }
788
789
790     /* set up all data structures needed for computation */
791     void Model::setup_dof ()
792     {
793         computing_timer.enter_section("Setup dof systems");
794
795         dof_handler.distribute_dofs (fe);
796
797         constraints.clear ();
798         DoFTools::make_hanging_node_constraints (dof_handler, constraints);
799         constraints.close ();
800
801         system_matrix.clear();
802
803         CompressedSparsityPattern c_sparsity (dof_handler.n_dofs());
804         DoFTools::make_sparsity_pattern (dof_handler, c_sparsity, constraints,
805                                          false);
806         sparsity_pattern.copy_from(c_sparsity);
807
808         system_matrix.reinit(sparsity_pattern);
809
810         solution.reinit (dof_handler.n_dofs());
811         old_solution.reinit(dof_handler.n_dofs());
812         system_rhs.reinit (dof_handler.n_dofs());
813
814         /* Optional: display sparsity of global matrix */
815         // std::ofstream out("sparsity_pattern.m2");
816         // sparsity_pattern.print_gnuplot(out);
817         computing_timer.exit_section();
818     }
819
820     /* compute the main matrix and right hand side */
821     void Model::assemble_term ()
822     {
823         computing_timer.enter_section ("   Assembling system");
824         system_rhs = 0;
825         compute_oldterm ();
826         rhs.advance_time (time_step);
827         compute_newterm ();
828         computing_timer.exit_section();
829     }
830
831
832     void Model::compute_oldterm ()
833     {
```

```
834         const QGauss<2> quadrature_formula (3);
835         FEValues<2>     fe_values (mapping, fe, quadrature_formula,
836                                    update_values | update_gradients |
837                                    update_JxW_values | update_q_points);
838
839         const unsigned int dofs_per_cell = dof_handler.get_fe().dofs_per_cell;
840         const unsigned int n_q_points    = quadrature_formula.size();
841         double coeff = 0.;
842         double cont = 1.;
843
844         Vector<double> local_term (dofs_per_cell);
845         std::vector<unsigned int> local_dof_indices (dofs_per_cell);
846         std::vector<double> old_data_values (n_q_points);
847         std::vector<Tensor<1,2> > old_data_grads (n_q_points);
848
849         std::vector<double> rhs_values_old (n_q_points);
850
851         DoFHandler<2>::active_cell_iterator
852         cell = dof_handler.begin_active(),
853         endc = dof_handler.end();
854
855         for(; cell!=endc; ++cell){
856                 local_term = 0;
857                 fe_values.reinit (cell);
858                 fe_values.get_function_values (old_solution, old_data_values);
859                 fe_values.get_function_grads (old_solution, old_data_grads);
860                 rhs.value_list(fe_values.get_quadrature_points(), rhs_values_old);
861
862         for(unsigned int q_point=0; q_point<n_q_points; ++q_point){
863                 coeff = compute_kstar(old_data_values[q_point], cell->material_id());
864
865            for(unsigned int i=0; i<dofs_per_cell; ++i){
866                 local_term(i) += (1/(time_step)*old_data_values[q_point]*
867                                   fe_values.shape_value(i,q_point) + (1-theta)*
868                                   rhs_values_old[q_point]*
869                                   fe_values.shape_value(i, q_point) - coeff*cont*
870                                   (1-theta)*old_data_grads[q_point]*
871                                   fe_values.shape_grad(i,q_point))*
872                                   fe_values.JxW (q_point);
873            }
874         }
875
876         cell->get_dof_indices (local_dof_indices);
877
878            for (unsigned int i=0; i<dofs_per_cell; ++i){
879                 system_rhs(local_dof_indices[i]) += local_term(i);
880            }
881         }
882         constraints.condense (system_rhs);
```

```
883   }
884
885   void Model::compute_newterm ()
886   {
887         const QGauss<2> quadrature_formula (3);
888         FEValues<2>     fe_values (mapping, fe, quadrature_formula,
889                                    update_values | update_gradients |
890                                    update_JxW_values | update_q_points);
891
892         const unsigned int dofs_per_cell = fe.dofs_per_cell;
893         const unsigned int n_q_points    = quadrature_formula.size();
894
895         Vector<double> local_term (dofs_per_cell);
896         std::vector<unsigned int> local_dof_indices (dofs_per_cell);
897
898         std::vector<double> rhs_values_new (n_q_points);
899
900         DoFHandler<2>::active_cell_iterator
901         cell = dof_handler.begin_active(),
902         endc = dof_handler.end();
903
904         for(; cell!=endc; ++cell){
905                 local_term = 0;
906                 fe_values.reinit (cell);
907                 rhs.value_list(fe_values.get_quadrature_points(), rhs_values_new);
908
909            for (unsigned int q_point=0; q_point<n_q_points; ++q_point){
910                 for (unsigned int i=0; i<dofs_per_cell; ++i){
911                         local_term(i) += theta*rhs_values_new[q_point]*
912                                          fe_values.shape_value(i,q_point)*
913                                          fe_values.JxW (q_point);
914                 }
915            }
916
917            cell->get_dof_indices (local_dof_indices);
918
919            for (unsigned int i=0; i<dofs_per_cell; ++i){
920                 system_rhs(local_dof_indices[i]) += local_term(i);
921            }
922
923         }
924         constraints.condense(system_rhs);
925   }
926
927   /* compute initial matrix */
928   void Model::compute_matrix ()
929   {
930         computing_timer.enter_section ("   Rebuilding matrix");
931
```

```
932         system_matrix = 0;
933
934         QGauss<2>   quadrature_formula (3);
935         FEValues<2> fe_values (mapping, fe, quadrature_formula,
936                                update_values | update_gradients | update_JxW_values |
937                                update_q_points);
938
939         const unsigned int dofs_per_cell = fe.dofs_per_cell;
940         const unsigned int n_q_points    = quadrature_formula.size();
941         double coeff = 0;
942         double cont = 1.;
943
944         FullMatrix<double> local_matrix (dofs_per_cell, dofs_per_cell);
945         std::vector<unsigned int> local_dof_indices (dofs_per_cell);
946         std::vector<double> old_data_values (n_q_points);
947
948         DoFHandler<2>::active_cell_iterator
949         cell = dof_handler.begin_active(),
950         endc = dof_handler.end();
951
952         for (; cell!=endc; ++cell){
953                 local_matrix = 0;
954                 fe_values.reinit (cell);
955                 fe_values.get_function_values (old_solution, old_data_values);
956
957           for (unsigned int q_point=0; q_point<n_q_points; ++q_point){
958                 coeff = compute_kstar(old_data_values[q_point], cell->material_id());
959              for (unsigned int i=0; i<dofs_per_cell; ++i){
960                 for (unsigned int j=0; j<dofs_per_cell; ++j){
961                         local_matrix(i,j) += (1/(time_step)*
962                                               fe_values.shape_value(i,q_point)*
963                                               fe_values.shape_value(j,q_point) +
964                                               theta*coeff*cont*
965                                               fe_values.shape_grad(i, q_point)*
966                                               fe_values.shape_grad(j, q_point))*
967                                               fe_values.JxW (q_point);
968                 }
969              }
970           }
971
972           cell->get_dof_indices (local_dof_indices);
973
974           for (unsigned int i=0; i<dofs_per_cell; ++i){
975               for (unsigned int j=0; j<dofs_per_cell; ++j){
976                   system_matrix.add(local_dof_indices[i], local_dof_indices[j],
977                                     local_matrix(i,j));
978               }
979           }
980         }
```

```
981
982         constraints.condense (system_matrix);
983         computing_timer.exit_section();
984     }
985
986
987     /* rebuilding matrix if necessary */
988     void Model::assemble_system ()
989     {
990           if(rebuild_matrix == true){
991                   compute_matrix();
992           }
993           assemble_term ();
994     }
995
996     /* solve the resulting system with a cg solver */
997     void Model::solve ()
998     {
999           computing_timer.enter_section ("   Solve system");
1000
1001          SolverControl solver_control (1000, 1e-12, false, false);
1002          SolverCG<> cg (solver_control);
1003          preconditioner.initialize(system_matrix, 1.2);
1004
1005          FilteredMatrix<Vector<double> > f_matrix(system_matrix);
1006          std::map<unsigned int,double> boundary_values;
1007
1008          if(SillData::geotherm == 0){
1009          VectorTools::interpolate_boundary_values (dof_handler,
1010                                                    0,
1011                                                    boundarydata1,
1012                                                    boundary_values);
1013          }
1014          else{
1015                  VectorTools::interpolate_boundary_values (dof_handler,
1016                                                            0,
1017                                                            boundarydata2,
1018                                                            boundary_values);
1019          }
1020
1021          f_matrix.add_constraints(boundary_values);
1022          f_matrix.apply_constraints(system_rhs, true);
1023          cg.solve (f_matrix, solution, system_rhs, preconditioner);
1024          constraints.distribute (solution);
1025          std::cout << solver_control.last_step()
1026                    << " CG iterations."
1027                    << std::endl;
1028          computing_timer.exit_section();
1029    }
```

```
1030
1031
1032    /* output results with gnuplot */
1033    void Model::output_results (const unsigned int timestep_number) const
1034    {
1035          DataOut<2> data_out_one;
1036          data_out_one.attach_dof_handler (dof_handler);
1037          data_out_one.add_data_vector (solution, "u");
1038          data_out_one.build_patches ();
1039          std::string name = "not_the_solution.";
1040
1041          switch (test){
1042                  case Parameters::TEST_ZERO:
1043                  name = "fsolution.TEST0.";
1044                  break;
1045
1046                  case Parameters::TEST_ONE:
1047                  name = "fsolution.TEST1.";
1048                  break;
1049
1050                  default:
1051                  Assert (false, ExcNotImplemented());
1052          };
1053
1054          std::string tname = "not";
1055
1056          tname = "0.";
1057
1058          const std::string filename1 = name + tname  +
1059                                        Utilities::int_to_string (timestep_number, 5) +
1060                                        ".gnuplot";
1061
1062          std::ofstream output_one (filename1.c_str());
1063          data_out_one.write_gnuplot (output_one);
1064    }
1065
1066
1067    /* evaluate numerical solution at a particular point */
1068    double Model::evaluate_soln (double x, double y) const
1069    {
1070          return VectorTools::point_value (mapping, dof_handler, solution,
1071                                           Point<2>(x, y));
1072    }
1073
1074    /* create initial uniform grid */
1075    void Model::make_initial_grid(){
1076          const Point<2> botleft = (Point<2> (0., 0.));
1077          const Point<2> upright = (Point<2> (SillData::DomainLength,
1078                                      SillData::DomainHeight));
```

```
1079
1080          std::vector<unsigned int> n_div;
1081          n_div.push_back(ini_refinement_level);
1082          n_div.push_back(ini_refinement_level);
1083
1084          GridGenerator::subdivided_hyper_rectangle(triangulation, n_div,
1085                                                    botleft, upright);
1086     }
1087
1088
1089    /* run the simulation in time */
1090    void Model::run ()
1091    {
1092       std::cout << "============================ Running TEST_" << test
1093                 << std::endl;
1094       std::cout << "============================ Geotherm = " << SillData::geotherm
1095                 << std::endl;
1096       std::cout << "Time step #0" << std::endl;
1097
1098       /* open file to read temperature solutions */
1099       std::ofstream file_one;
1100
1101       std::string name, tname, tmode;
1102       if(test == Parameters::TEST_ZERO){
1103           name = "T0_";
1104       }
1105       else{
1106           name = "T1_";
1107       }
1108
1109       tname = "0_";
1110
1111       std::string filename1 = name + tname  + "soln.txt";
1112
1113       file_one.open(filename1.c_str());
1114
1115       make_initial_grid();
1116
1117       if(mode == 1){
1118           triangulation.refine_global(level);
1119       }
1120
1121       setup_dof();
1122
1123       unsigned int pre_refinement_step = 0;
1124
1125       start_time_iteration:
1126
1127       time = 0.;
```

```
1128      unsigned int timestep_number = 1;
1129
1130      /* assign material id to each cell */
1131      if(test != Parameters::TEST_ZERO){
1132          for (Triangulation<2>::active_cell_iterator
1133               cell = triangulation.begin_active();
1134               cell!=triangulation.end();
1135               ++cell){
1136                  if(cell->center()[0] <= SillData::L2 && cell->center()[0]
1137                          >= SillData::L1 && cell->center()[1] <= SillData::W2 &&
1138                          cell->center()[1] >= SillData::W1){
1139                          cell->recursively_set_material_id('m');
1140                   }
1141                   else{
1142                          cell->recursively_set_material_id('c');
1143                  }
1144               }
1145      }
1146
1147      compute_matrix();
1148
1149      /* project initial conditions onto initial mesh */
1150      if(SillData::geotherm == 0){
1151          VectorTools::project (mapping, dof_handler,
1152                  constraints,
1153                  QGauss<2>(4),
1154                  InitialValues_ZeroGeotherm (1, time),
1155                  solution);
1156      }
1157      else{
1158           VectorTools::project (mapping, dof_handler,
1159                  constraints,
1160                  QGauss<2>(4),
1161                  InitialValues_Geotherm (1, time),
1162                  solution);
1163     }
1164
1165
1166
1167     if(mode != 1 && timestep_number == 1 &&
1168          pre_refinement_step < n_pre_refinement_steps){
1169                  refine_grid(n_pre_refinement_steps);
1170                  ++pre_refinement_step;
1171                  goto start_time_iteration;
1172     }
1173
1174     output_results (0);
1175     /* loop over time to compute temperature solutions */
1176     for(time+=time_step; time<=final_time; time+=time_step, ++timestep_number){
```

```
1177
1178          std::cout << "Time step dt = " << time_step << std::endl;
1179
1180          std::cout << "Number of active cells: "
1181          << triangulation.n_active_cells()
1182          << std::endl
1183          << "Total number of cells: "
1184          << triangulation.n_cells()
1185          << std::endl
1186          << "Total dofs: " << dof_handler.n_dofs()
1187          << std::endl;
1188
1189          /* adaptive mesh refinement every "freq" time steps */
1190          if(mode != 1 && (timestep_number % freq == 0)){
1191                  refine_grid(n_pre_refinement_steps);
1192          }
1193
1194          if(SillData::geotherm == 0){
1195                  boundarydata1.advance_time(time_step);
1196          }
1197          else{
1198                  boundarydata2.advance_time(time_step);
1199          }
1200
1201          if(timestep_number == 1){
1202                  file_one << timestep_number-1 << " "
1203                           << dof_handler.n_dofs()
1204                           << " " << evaluate_soln(1.25,0.525)
1205                           << " " << evaluate_soln(1.25,0.530)
1206                           << " " << evaluate_soln(1.5,0.525)
1207                           << " " << evaluate_soln(1.5,0.530)
1208                           << std::endl;
1209          }
1210
1211          old_solution = solution;
1212
1213          std::cout << std::endl
1214                    << "Time step #" << timestep_number << "; "
1215                    << "advancing to t = " << time << "."
1216                    << std::endl;
1217
1218          assemble_system ();
1219
1220          solve ();
1221
1222          output_results (timestep_number);
1223
1224          if(timestep_number >= 1){
1225                  file_one << timestep_number
```

```
1226                          << " " << dof_handler.n_dofs()
1227                          << " " << evaluate_soln(1.25,0.525)
1228                          << " " << evaluate_soln(1.25,0.530)
1229                          << " " << evaluate_soln(1.5,0.525)
1230                          << " " << evaluate_soln(1.5,0.530)
1231                          << std::endl;
1232
1233          }
1234
1235          /* stop the program when the entire crust domain solidifies */
1236          if(evaluate_soln((SillData::L1 + SillData::L2)/2,
1237                  (SillData::W1 + SillData::W2)/2 ) < TM_0){
1238                          summary.open("Timer.txt");
1239                          computing_timer.print_summary ();
1240                          summary.close();
1241                          file_one << "Solidification time : " << time << std::endl;
1242                          file_one.close();
1243                          return;
1244          }
1245      }
1246
1247    }
1248  /* end of Earth namespace */
1249  }
1250
1251
1252  int main ()
1253  {
1254    using namespace dealii;
1255    using namespace Earth;
1256
1257    try
1258    {
1259      /* read inputs from parameter.prm */
1260      Parameters::DataInput data;
1261      data.read_data ("parameter.prm");
1262
1263      deallog.depth_console (0);
1264
1265      Model model(data);
1266      model.run ();
1267    }
1268    catch (std::exception &exc)
1269     {
1270      std::cerr << std::endl << std::endl
1271        << "----------------------------------------------------"
1272        << std::endl;
1273      std::cerr << "Exception on processing: " << std::endl
1274        << exc.what() << std::endl
```

```
1275        << "Aborting!" << std::endl
1276        << "----------------------------------------------------"
1277        << std::endl;
1278
1279      return 1;
1280    }
1281    catch (...)
1282    {
1283      std::cerr << std::endl << std::endl
1284        << "----------------------------------------------------"
1285        << std::endl;
1286      std::cerr << "Unknown exception!" << std::endl
1287        << "Aborting!" << std::endl
1288        << "----------------------------------------------------"
1289        << std::endl;
1290      return 1;
1291    }
1292
1293    return 0;
1294  }
1295
1296  ==================================== Makefile ============================================
1297  # $Id: Makefile 24349 2011-09-21 08:38:41Z kronbichler $
1298
1299  # For the small projects Makefile, you basically need to fill in only
1300  # four fields.
1301  #
1302  # The first is the name of the application. It is assumed that the
1303  # application name is the same as the base file name of the single C++
1304  # file from which the application is generated.
1305      target = source
1306
1307  # The second field determines whether you want to run your program in
1308  # debug or optimized mode. The latter is significantly faster, but no
1309  # run-time checking of parameters and internal states is performed, so
1310  # you should set this value to ‘on’ while you develop your program,
1311  # and to ‘off’ when running production computations.
1312  debug-mode = on
1313
1314  # As third field, we need to give the path to the top-level deal.II
1315  # directory. You need to adjust this to your needs. Since this path is
1316  # probably the most often needed one in the Makefile internals, it is
1317  # designated by a single-character variable, since that can be
1318  # reference using $D only, i.e. without the parentheses that are
1319  # required for most other parameters, as e.g. in $(target).
1320  D = ../../
1321
1322  # The last field specifies the names of data and other files that
1323  # shall be deleted when calling ‘make clean’. Object and backup files,
```

```
1324 # executables and the like are removed anyway. Here, we give a list of
1325 # files in the various output formats that deal.II supports.
1326 clean-up-files = *gmv *gnuplot *gpl *eps *pov *vtk *ucd *.d2
1327
1328 #
1329 #
1330 # Usually, you will not need to change anything beyond this point.
1331 #
1332 #
1333 # The next statement tells the ‘make’ program where to find the
1334 # deal.II top level directory and to include the file with the global
1335 # settings
1336 include $D/common/Make.global_options
1337
1338 # Since the whole project consists of only one file, we need not
1339 # consider difficult dependencies. We only have to declare the
1340 # libraries which we want to link to the object file. deal.II has two
1341 # libraries: one for the debug mode version of the
1342 # application and one for optimized mode.
1343 libs.g   := $(lib-deal2.g)
1344 libs.o   := $(lib-deal2.o)
1345
1346 # We now use the variable defined above to switch between debug and
1347 # optimized mode to select the set of libraries to link with. Included
1348 # in the list of libraries is the name of the object file which we
1349 # will produce from the single C++ file. Note that by default we use
1350 # the extension .g.o for object files compiled in debug mode and .o for
1351 # object files in optimized mode (or whatever local default on your
1352 # system is instead of .o)
1353 ifeq ($(debug-mode),on)
1354   libraries = $(target).g.$(OBJEXT) $(libs.g)
1355 else
1356   libraries = $(target).$(OBJEXT) $(libs.o)
1357 endif
1358
1359 # Now comes the first production rule: how to link the single object
1360 # file produced from the single C++ file into the executable. Since
1361 # this is the first rule in the Makefile, it is the one ‘make’ selects
1362 # if you call it without arguments.
1363 $(target)$(EXEEXT) : $(libraries)
1364 @echo ============================ Linking $@
1365 @$(CXX) -o $@ $^ $(LIBS) $(LDFLAGS)
1366
1367 # To make running the application somewhat independent of the actual
1368 # program name, we usually declare a rule ‘run’ which simply runs the
1369 # program. You can then run it by typing ‘make run’. This is also
1370 # useful if you want to call the executable with arguments which do
1371 # not change frequently. You may then want to add them to the
1372 # following rule:
```

```
1373 run: $(target)$(EXEEXT)
1374 @echo ============================ Running $<
1375 @./$(target)$(EXEEXT)
1376
1377 # As a last rule to the ‘make’ program, we define what to do when
1378 # cleaning up a directory. This usually involves deleting object files
1379 # and other automatically created files such as the executable itself,
1380 # backup files, and data files. Since the latter are not usually quite
1381 # diverse, you needed to declare them at the top of this file.
1382 clean:
1383 -rm -f *.$(OBJEXT) *~ Makefile.dep $(target)$(EXEEXT) $(clean-up-files)
1384
1385 # Since we have not yet stated how to make an object file from a C++
1386 # file, we should do so now. Since the many flags passed to the
1387 # compiler are usually not of much interest, we suppress the actual
1388 # command line using the ‘at’ sign in the first column of the rules
1389 # and write the string indicating what we do instead.
1390 ./%.g.$(OBJEXT) :
1391 @echo "==============debug========= $(<F)  ->  $@"
1392 @$(CXX) $(CXXFLAGS.g) -c $< -o $@
1393 ./%.$(OBJEXT) :
1394 @echo "==============optimized===== $(<F)  ->  $@"
1395 @$(CXX) $(CXXFLAGS.o) -c $< -o $@
1396
1397 # The following statement tells make that the rules ‘run’ and ‘clean’
1398 # are not expected to produce files of the same name as Makefile rules
1399 # usually do.
1400 .PHONY: run clean
1401
1402
1403 # Finally there is a rule which you normally need not care much about:
1404 # since the executable depends on some include files from the library,
1405 # besides the C++ application file of course, it is necessary to
1406 # re-generate the executable when one of the files it depends on has
1407 # changed. The following rule creates a dependency file
1408 # ‘Makefile.dep’, which ‘make’ uses to determine when to regenerate
1409 # the executable. This file is automagically remade whenever needed,
1410 # i.e. whenever one of the cc-/h-files changed. Make detects whether
1411 # to remake this file upon inclusion at the bottom of this file.
1412 #
1413 # If the creation of Makefile.dep fails, blow it away and fail
1414 Makefile.dep: $(target).cc Makefile \
1415               $(shell echo $D/include/deal.II/*/*.h)
1416 @echo ============================ Remaking $@
1417 @$D/common/scripts/make_dependencies  $(INCLUDE) -B. $(target).cc \
1418                 > $@ \
1419           || (rm -f $@ ; false)
1420 @if test -s $@ ; then : else rm $@ ; fi
1421
```

```
1422 # To make the dependencies known to ‘make’, we finally have to include
1423 # them:
1424 include Makefile.dep
```