IMPLEMENTATION OF ISP

ALI HAQVERDIYEV AND JONATHAN GOLDFARB

Abstract. TBD

Key words and phrases. TBD

Contents

1. Introduction

sec:inverseestefanproblem

1.1. Inverse Stefan Problem. Consider the general one-phase Stefan problem:

eq:intro-pde
$$Lu \equiv (a(t)b(x)u_x)_x - u_t = f, \text{ in } \Omega \tag{1}$$
eq:intro-init
$$u(x,0) = \phi(x),\ 0 \le x \le s(0) = s_0 \tag{2}$$
eq:pde-bound
$$a(t)b(0)u_x(0,t) = g(t),\ 0 \le t \le T \tag{3}$$
eq:pde-stefan
$$a(t)b(s(t))u_x(s(t),t) + \gamma(s(t),t)s'(t) = \chi(s(t),t),\ 0 \le t \le T \tag{4}$$
eq:pde-freebound
$$u(s(t),t) = \mu(t),\ 0 \le t \le T, \tag{5}$$

Date: March 15, 2019.

2010 *Mathematics Subject Classification.* TBD.

where

eq:pde-domain-defn
$$\Omega = \{(x,t) : 0 < x < s(t),\ 0 < t \le T\} \tag{6}$$

where a, b, f, ϕ, g, γ, χ, μ are given functions. Assume now that a is unknown; in order to find a, along with u and s, we must have additional information. Assume that we are able to measure the temperature on our domain and the position of the free boundary at the final moment T.

eq:pde-finaltemp
$$u(x,T) = w(x),\ 0 \le x \le s(T) = s_*. \tag{7}$$

Under these conditions, we are required to solve an *inverse* Stefan problem (ISP): find a tuple

$$\{u(x,t), s(t), a(t)\}$$

that satisfy conditions (1)–(7). ISP is not well posed in the sense of Hadamard: the solution may not exist; if it exists, it may not be unique, and in general it does not exhibit continuous dependence on the data. The main methods available for ISP are based on a variational formulation, Frechet differentiability, and iterative gradient methods. We cite recent papers [?, ?] and the monograph [?] for a list of references. The established variational methods in earlier works fail in general to address two issues:

- The solution of ISP does not depend continuously on the phase transition temperature $\mu(t)$ from (5). A small perturbation of the phase transition temperature may imply significant change of the solution to the inverse Stefan problem.
- In the existing formulation, at each step of the iterative method a Stefan problem must be solved (that is, for instance, the unknown heat flux g is given, and the corresponding u and s are calculated) which incurs a high computational cost.

A new method developed in [?, ?] addresses both issues with a new variational formulation. The key insight is that the free boundary problem has a similar nature to an inverse problem, so putting them into the same framework gives a conceptually clear formulation of the problem; proving existence, approximation, and differentiability is a resulting challenge. Existence of the optimal control and the convergence of the sequence of discrete optimal control problems to the continuous optimal control problem was proved in [?, ?]. In [?, ?], Frechet differentiability of the new variational formulation was developed, and a full result showing Frechet differentiability and the form of the Frechet differential with respect to the free boundary, sources, and coefficients were proven. Our goal in this work is to document the implementation in MATLAB using the built-in PDEPE solver.

2. Forward Problem for ISP

sec:forward-problem

We will consider the problem (1)–(4) where $(s,a) \in V_R$ is fixed, and

eq:control-set
$$\begin{aligned} V_R = \Big\{ v = (s,a) \in H, |v|_H \le R;\ s(0) = s_0,\ g(0) = a(0)b(0)\phi'(0),\ a_0 \le a(t) \\ \chi(s_0,0) = \phi'(s_0)a(0)b(s_0) + \gamma(s_0,0)s'(0),\ 0 < \delta \le s(t) \Big\}, \end{aligned} \tag{8}$$

where $\beta_0, \beta_1, \beta_2 \ge 0$ and $a_0, \delta, R > 0$ are given, and

$$H := W_2^1(0,T) \times W_2^2(0,T)$$
$$\|v\|_H := \max\Big(\|a\|_{W_2^1(0,T)}, \|s\|_{W_2^2(0,T)} \Big)$$

Define

$$D := \{(x,t) : 0 \le x \le \ell,\ 0 \le t \le T\},$$

where $l = l(R) > 0$ is chosen such that for any control $v \in V_R$, its component s satisfies $s(t) \leq l$. For a given control vector $v = (s, a) \in V_R$ transform the domain Ω to the cylindrical domain Q_T by the change of variables $y = x/s(t)$. Let $d = d(x,t)$, $(x,t) \in \Omega$ stand for any of u, b, f, γ, χ, define the function $\tilde{d}$ by

$$\tilde{d}(x,t) = d\big(xs(t),t\big),\ \tilde{b}(x) = b(xs(t)),\ \text{and } \tilde{\phi}(x) = \phi\big(xs_0\big)$$

The transformed function $\tilde{u}$ is a ***pointwise a.e.*** solution of the Neumann problem

eq:tform-pde
$$\frac{a}{s^2}\big(\tilde{b}\tilde{u}_y\big)_y + \frac{ys'}{s}\tilde{u}_y - \tilde{u}_t = \tilde{f},\ \text{in } Q_T \tag{9}$$

eq:tform-iv
$$\tilde{u}(x,0) = \tilde{\phi}(x),\ 0 \leq x \leq 1 \tag{10}$$

eq:tform-lbdy
$$a(t)b(0)\tilde{u}_y(0,t) = g(t)s(t),\ 0 \leq t \leq T \tag{11}$$

eq:tform-rbdy
$$a(t)b(1)\tilde{u}_y(1,t) = \tilde{\chi}(1,t)s(t) - \tilde{\gamma}(1,t)s'(t)s(t),\ 0 \leq t \leq T \tag{12}$$

2.1. **MATLAB Implementation of Forward Problem**. The built-in MATLAB PDE solver, `pdepe`, solves parabolic-elliptic problems in one space dimension. In particular, it uses a method-of-lines technique as a differential-algebraic equation solver applied to problems of the form

eq:matlab-pde
$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m F\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + d\left(x,t,u,\frac{\partial u}{\partial x}\right) \tag{13}$$

for

$$a \leq x \leq b, \quad t_0 \leq t \leq t_f \tag{14}$$

where $m = 0,1,2$ is fixed, c is a diagonal matrix of size $n \times n$, where there are n components in u. There must be at least one parabolic equation, which corresponds to the condition of at least one component of c being positive. The solution components satisfy

eq:matlab-ic
$$u(x,t_0) = u_0(x) \tag{15}$$

and boundary conditions of the form

eq:matlab-bc
$$p(x,t,u) + q(x,t)F\left(x,t,u,\frac{\partial u}{\partial x}\right)\Big|_{x=a,b} = 0 \tag{16}$$

In particular, the function F appearing in the boundary condition is the same as the flux term in the PDE. In the MATLAB implementation, the method `Forward` is intended to produce a function $u(x,t)$ by computing $\tilde{u}(y,t)$ and returning $u(x,t) = \tilde{u}(x/s(t),t)$. The subfunction `pdeSolver` handles the conversion of problems of the form (9)–(12) to the form that the MATLAB solver requires.

We first write system (9)–(12) in the form

$$s^2\tilde{u}_t = a\big(\tilde{b}\tilde{u}_y\big)_y + sys'\tilde{u}_y - s^2\tilde{f},\ \text{in } Q_T =: (0,1)\times(0,T) \tag{17}$$

$$\tilde{u}(x,0) = \tilde{\phi}(x),\ 0 \leq x \leq 1 \tag{18}$$

$$a(t)b(0)\tilde{u}_y(0,t) = g(t)s(t),\ 0 \leq t \leq T \tag{19}$$

$$a(t)\tilde{b}(1)\tilde{u}_y(1,t) = \tilde{\chi}(1,t)s(t) - \tilde{\gamma}(1,t)s'(t)s(t),\ 0 \leq t \leq T \tag{20}$$

Evidently, we must take $a = 0$, $b = 1$, $t_0 = 0$, $t_f = T$, and $m = 0$; then, identifying the diffusion coefficient between (9) and (13), we observe that

$$F(x,t,u,r) := a(t)\tilde{b}(x)r$$

and by considering the time derivative, evidently

$$c(x,t,u,r) := s^2(t)$$

and lastly,

$$d(x,t,u,r) := s(t)\big(s'(t)xr - s(t)\tilde{f}(x,t)\big)$$

The initial data should be taken as $u_0(x) = \tilde{\phi}(x) = \phi(xs(t))$, and the boundary conditions take the form

$$a(t)\tilde{b}(0)\tilde{u}_y(0,t) \equiv F(0,t,\tilde{u},\tilde{u}_y) = p(0,t,\tilde{u}) := g(t)s(t),$$
$$q(0,t) = 1$$
$$a(t)\tilde{b}(1)\tilde{u}_y(1,t) \equiv F(1,t,\tilde{u},\tilde{u}_y) = p(1,t,\tilde{u}) := s(t)\left(\tilde{\chi}(1,t) - \tilde{\gamma}(1,t)s'(t)\right)$$

2.2. **Formulation of Reduced Model for Forward Problem**.

Consider the problem (1)–(4) with $\gamma \equiv 1$, $\chi \equiv 0$, and $b \equiv 1$:

eq:reduced-pde
$$Lu \equiv a(t)u_{xx} - u_t = f,\ \text{in } \Omega \tag{21}$$
eq:reduced-init
$$u(x,0) = \phi(x),\ 0 \le x \le s(0) = s_0 \tag{22}$$
eq:reduced-lbc
$$a(t)u_x(0,t) = g(t),\ 0 \le t \le T \tag{23}$$
eq:reduced-stefan
$$a(t)u_x(s(t),t) + s'(t) = 0,\ 0 \le t \le T \tag{24}$$

The corresponding problem (9)–(12) reduces to to give

$$s^2\tilde{u}_t = a\big(\tilde{u}_y\big)_y + sys'\tilde{u}_y - s^2\tilde{f},\ \text{in } Q_T \tag{25}$$
$$\tilde{u}(y,0) = \tilde{\phi}(y),\ 0 \le x \le 1 \tag{26}$$
$$a(t)\tilde{u}_y(0,t) = g(t)s(t),\ 0 \le t \le T \tag{27}$$
$$a(t)\tilde{u}_y(1,t) = -s'(t)s(t),\ 0 \le t \le T \tag{28}$$

and the mapping from the notation in our paper to the notation preferred by MATLAB becomes

$$F(x,t,u,r) := a(t)r$$
$$c(x,t,u,r) := s^2(t)$$
$$d(x,t,u,r) := s(t)\big(s'(t)xr - s(t)\tilde{f}(x,t)\big)$$

The initial data should be taken as $u_0(x) = \tilde{\phi}(x) = \phi(xs(t))$, and the boundary conditions take the form

$$a(t)\tilde{u}_y(0,t) \equiv F(0,t,\tilde{u},\tilde{u}_y) = p(0,t,\tilde{u}) := g(t)s(t),$$
$$a(t)\tilde{u}_y(1,t) \equiv F(1,t,\tilde{u},\tilde{u}_y) = p(1,t,\tilde{u}) := -s(t)s'(t)$$

and

$$q(x,t) \equiv 1$$

Under the additional assumption that the sources $f \equiv 0$, the mapping simplifies further to

$$F(x,t,u,r) := a(t)r$$
$$c(x,t,u,r) := s^2(t)$$
$$d(x,t,u,r) := s(t)s'(t)xr$$

The boundary conditions take the same form with the updated definition of F. This is the model implemented in `Forward.m` (specifically, in the `pdeSolver` subfunction), included in Section A.1. The model problem currently implemented in `test_Forward.m`, included in Section A.2 can be found in Section **??**.

3. Optimal Control Problem and Gradient-Based Approach

Consider the minimization of the functional

eq:functional
$$\mathcal{J}(v) = \beta_0 \int_0^{s(T)} |u(x,T;v) - w(x)|^2 \, dx + \beta_1 \int_0^T |u(s(t),t;v) - \mu(t)|^2 \, dt + \beta_2 |s(T) - s_*|^2 \tag{29}$$

on the control set (8). The formulated optimal control problem (8), (9)–(12), (29) will be called Problem $\mathcal{I}$.

defn:adjoint

Definition 1. For given v and $u = u(x,t;v)$, ψ is a solution to the adjoint problem if

eq:adj-pde
$$L^*\psi := \big(ab\psi_x\big)_x + \psi_t = 0, \quad \text{in } \Omega \tag{30}$$

eq:adj-finalmoment
$$\psi(x,T) = 2\beta_0(u(x,T) - w(x)), \ 0 \le x \le s(T) \tag{31}$$

eq:adj-robin-fixed
$$b(0)a(t)\psi_x(0,t) = 0, \ 0 \le t \le T \tag{32}$$

eq:adj-robin-free
$$\Big[ab\psi_x - s'\psi - 2\beta_1(u-\mu)\Big]_{x=s(t)} = 0, \ 0 \le t \le T \tag{33}$$

thm:gradient-result

Theorem 1. *Problem $\mathcal{I}$ has a solution, and the functional $\mathcal{J}(v)$ is differentiable in the sense of Frechet, and the first variation is*

eq:gradient-full
$$\begin{aligned}\langle \mathcal{J}'(v), \Delta v\rangle_H &= \int_0^T \Big[2\beta_1\big(u-\mu\big)u_x + \psi\big(\chi_x - \gamma_x s' - a(bu_x)_x\big)\Big]_{x=s(t)} \Delta s(t)\, dt \\ &+ \Big[\beta_0 |u(s(T),T) - w(s(T))|^2 + 2\beta_2(s(T) - s_*)\Big] \Delta s(T) - \int_0^T \big[\psi\gamma\big]_{x=s(t)} \Delta s'(t)\, dt \\ &+ \int_0^T \Delta a \left[\int_0^{s(t)} (bu_x)_x \psi \, dx - [bu_x\psi]_{x=s(t)} - [bu_x\psi]_{x=0}\right] dt\end{aligned} \tag{34}$$

where $\mathcal{J}'(v) \in H'$ is the Frechet derivative, $\langle\cdot,\cdot\rangle_H$ is a pairing between H and its dual H', ψ is a solution to the adjoint problem in the sense of definition 1, and $\delta v = (\Delta s, \Delta a)$ is a variation of the control vector $v \in V_R$ such that $v + \delta v \in V_R$.

3.1. MATLAB Implementation of Adjoint Problem

3.1. MATLAB Implementation of Adjoint Problem. To implement the adjoint problem, we first change variables in eqs. (30)–(33) as $x \mapsto x/s(t)$, $t \mapsto T - t$ to derive

$$a(T-t)\big(b(ys(T-t))\tilde{\psi}_y\big)_y + s(T-t)ys'(T-t)\tilde{\psi}_y - s^2(T-t)\tilde{\psi}_t = 0, \quad \text{in } Q_T := (0,1)\times(0,T) \tag{35}$$

$$\tilde{\psi}(y,0) = 2\beta_0(u(ys(T),T) - w(ys(T))), \ 0 \le y \le 1 \tag{36}$$

$$b(0)a(T-t)\tilde{\psi}_y(0,T-t) = 0, \ 0 \le t \le T \tag{37}$$

$$a(T-t)b(s(T-t))\tilde{\psi}_y(1,t) = -s(T-t)s'(T-t)\tilde{\psi}(1,t) + 2\beta_1 s(T-t)(u(s(T-t),t) - \mu(T-t)) = 0, \ 0 \le t \le T \tag{38}$$

Comparing with the notation of eqs. (13)–(16), we see that once again we must take $a = 0$, $b = 1$, $t_0 = 0$, $t_f = T$, and $m = 0$; hence (13) has the form

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}F\left(x,t,u,\frac{\partial u}{\partial x}\right) + d\left(x,t,u,\frac{\partial u}{\partial x}\right)$$

so

$$F(x,t,u,r) := a(T-t)b(ys(T-t))r$$
$$c(x,t,u,r) := s^2(T-t)$$
$$d(x,t,u,r) := xs(T-t)s'(T-t)r$$

The initial condition (15) has the form

$$\tilde{\psi}(x,0) = 2\beta_0(u(xs(T),T) - w(xs(T)))$$

The boundary conditions have the form

$$\begin{gathered} b(0)a(T-t)\psi_y(0,t) \equiv F(0,t,\psi,\psi_y(0,t)) = 0 \equiv p(0,t,\psi) \\ a(T-t)b(s(T-t))\tilde{\psi}_y(1,t) \equiv F(1,t,\psi,\psi_y(1,t)) = p\big(1,t,\psi(1,t)\big), \\ p\big(1,t,r\big) := -s(T-t)s'(T-t)r + 2\beta_1 s(T-t)(u(s(T-t),t) - \mu(T-t)) \end{gathered}$$

3.2. **Formulation of Reduced Model for Adjoint Problem**. In the reduced model, we assume $\gamma \equiv$, $\chi \equiv 0$, and $b \equiv 1$, so the adjoint problem becomes

$$\big(a\psi_x\big)_x + \psi_t = 0, \quad \text{in } \Omega \tag{39}$$
$$\psi(x,T) = 2\beta_0(u(x,T) - w(x)),\ 0 \le x \le s(T) \tag{40}$$
$$a(t)\psi_x(0,t) = 0,\ 0 \le t \le T \tag{41}$$
$$\Big[a\psi_x - s'\psi - 2\beta_1(u-\mu)\Big]_{x=s(t)} = 0,\ 0 \le t \le T \tag{42}$$

and the transformed problem is

$$a(T-t)\tilde{\psi}_{yy} + s(T-t)ys'(T-t)\tilde{\psi}_y - s^2(T-t)\tilde{\psi}_t = 0, \quad \text{in } Q_T := (0,1)\times(0,T) \tag{43}$$
$$\tilde{\psi}(y,0) = 2\beta_0(u(ys(T),T) - w(ys(T))),\ 0 \le y \le 1 \tag{44}$$
$$a(T-t)\tilde{\psi}_y(0,T-t) = 0,\ 0 \le t \le T \tag{45}$$
(46)
$$a(T-t)\tilde{\psi}_y(1,t) = -s(T-t)s'(T-t)\tilde{\psi}(1,t) + 2\beta_1 s(T-t)(u(s(T-t),t) - \mu(T-t)) = 0,\ 0 \le t \le T$$

Therefore, corresponding parameter functions for the MATLAB interface take the form

$$F(x,t,u,r) := a(T-t)r$$
$$c(x,t,u,r) := s^2(T-t)$$
$$d(x,t,u,r) := xs(T-t)s'(T-t)r$$

and the boundary conditions remain the same after updating the definition of F.

This model is implemented in `Adjoint.m` (specifically, in the `pdeSolver` subfunction), included in Section A.3. The development of a reasonable formulation for a test problem for the adjoint solver is a work-in-progress; for now, the same problem is used in `test_Forward.m` and in `test_Adjoint.m` (taking the appropriate boundary measurements; see Section A.4) and we simply check that the adjoint problem vanishes as the measurement error vanishes.

3.3. **Common PDE Solver for Forward and Adjoint Problem**. Comparing the parameter functions for the reduced Forward problem,

$$\begin{gathered}F(x,t,u,r) := a(t)r\\ c(x,t,u,r) := s^2(t)\\ d(x,t,u,r) := s(t)s'(t)xr\\ q(x,t) := 1\\ p(0,t,r) := g(t)s(t)\\ p(1,t,r) := -s(t)s'(t)\end{gathered}$$

and the reduced Adjoint problem,

$$\begin{gathered}F(x,t,u,r) := a(T-t)r\\ c(x,t,u,r) := s^2(T-t)\\ d(x,t,u,r) := ys(T-t)s'(T-t)r\\ q(x,t) := 1\\ p(0,t,r) := 0\\ p(1,t,r) := -s(T-t)s'(T-t)r + 2\beta_1 s(T-t)(u(s(T-t),t) - \mu(T-t))\end{gathered}$$

we see that both problems have the same fundamental structure; exploiting this to reduce code duplication is a future project.

3.4. **Implementation of Gradient Formulae in MATLAB**. The last step in the implementation is to write the gradient update formula in MATLAB notation. For the gradient with respect to $x = s(t)$, we have

$$\begin{gathered}\int_0^T \Big[2\beta_1\big(u-\mu\big)u_x + \psi\big(\chi_x - \gamma_x s' - a(bu_x)_x\big)\Big]_{x=s(t)} \Delta s(t)\,dt\\ + \Big[\beta_0\left|u(s(T),T) - w(s(T))\right|^2 + 2\beta_2(s(T)-s_*)\Big]\,\Delta s(T) - \int_0^T \left[\psi\gamma\right]_{x=s(t)} \Delta s'(t)\,dt\end{gathered}$$

Under the assumptions used to formulate "reduced" models above ($\gamma \equiv 1$, $\chi \equiv 0$, $b \equiv 1$,) we have

$$\begin{gathered}\int_0^T \Big[2\beta_1\big(u-\mu\big)u_x - \psi a u_{xx}\Big]_{x=s(t)} \Delta s(t)\,dt\\ + \Big[\beta_0\left|u(s(T),T) - w(s(T))\right|^2 + 2\beta_2(s(T)-s_*)\Big]\,\Delta s(T) - \int_0^T \psi(s(t),t)\Delta s'(t)\,dt\end{gathered}$$

Pursuing integration by parts with respect to time, it follows that the corresponding gradient term is

$$\begin{gathered}\int_0^T \Big[2\beta_1\big(u-\mu\big)u_x - \psi a u_{xx} + \psi_x(s(t),t)s'(t) + \psi_t(s(t),t)\Big]_{x=s(t)} \Delta s(t)\,dt\\ + \Big[\beta_0\left|u(s(T),T) - w(s(T))\right|^2 + 2\beta_2(s(T)-s_*) - \psi(s(T),T)\Big]\,\Delta s(T)\end{gathered}$$

This is the routine implemented in `grad_s.m`, included in Section A.5.

4. Model Problem #1

sec:model-problem-1

As a model problem, we will take (1)–(4) with $b \equiv 1$, $\chi \equiv 0$, and $\gamma \equiv 1$:

$$Lu \equiv (a(t)u_x)_x - u_t = f, \text{ in } \Omega \tag{47}$$

$$u(x,0) = \phi(x),\ 0 \le x \le s(0) = s_0 \tag{48}$$

$$a(t)u_x(0,t) = g(t),\ 0 \le t \le T \tag{49}$$

$$a(t)u_x(s(t),t) + s'(t) = 0,\ 0 \le t \le T \tag{50}$$

In this case, the adjoint problem (30)–(33) then takes the form

$$L^*\psi := \big(a\psi_x\big)_x + \psi_t = 0, \quad \text{in } \Omega \tag{51}$$

$$\psi(x,T) = 2\beta_0(u(x,T) - w(x)),\ 0 \le x \le s(T) \tag{52}$$

$$a(t)\psi_x(0,t) = 0,\ 0 \le t \le T \tag{53}$$

$$\Big[a\psi_x - s'\psi - 2\beta_1(u-\mu)\Big]_{x=s(t)} = 0,\ 0 \le t \le T \tag{54}$$

and the gradient is

$$\begin{aligned}\langle \mathcal{J}'(v), \Delta v\rangle_H &= \int_0^T \Big[2\beta_1\big(u-\mu\big)u_x - a\psi u_{xx}\Big]_{x=s(t)} \Delta s(t)\,dt \\ &+ \Big[\beta_0\left|u(s(T),T) - w(s(T))\right|^2 + 2\beta_2(s(T) - s_*)\Big]\Delta s(T) - \int_0^T \psi\big|_{x=s(t)}\Delta s'(t)\,dt \\ &+ \int_0^T \Delta a\left[\int_0^{s(t)} u_{xx}\psi\,dx - [u_x\psi]_{x=0} - [u_x\psi]_{x=s(t)}\right]dt\end{aligned} \tag{55}$$

The model problem originates in the following: find $\big(u,\xi\big)$ satisfying

eq:tikhonov-samarskii-pde-1
$$\frac{\partial u_1}{\partial t} - k_1\frac{\partial^2 u_1}{\partial x^2} = 0,\ 0 < x < \xi(t),\ t > 0 \tag{56}$$

eq:tikhonov-samarskii-pde-2
$$\frac{\partial u_2}{\partial t} - k_2\frac{\partial^2 u_2}{\partial x^2} = 0,\ \xi(t) < x < \infty,\ t > 0 \tag{57}$$

$$u_1(0,t) = c_1,\ t > 0 \tag{58}$$

$$u_2(x,0) = c_2,\ x > \xi(0) \tag{59}$$

$$u_1(x,0) = c_1,\ x < \xi(0) \tag{60}$$

$$u_1(\xi(t),t) = u_2(\xi(t),t) = 0,\ t > 0 \tag{61}$$

khonov-samarskii-stefancond
$$k_1\frac{\partial u_1}{\partial x}(\xi(t),t) - k_2\frac{\partial u_2}{\partial x}(\xi(t),t) = \gamma\frac{d\xi}{dt}(t) \tag{62}$$

where $k_1, k_2, \gamma > 0$ are given, and without loss of generality $c_1 < 0$, $c_2 \ge 0$. Direct calculation verifies that the exact solution to (47)–(53) is

$$u_1(x,t) = c_1 + B_1\mathrm{erf}\left(\frac{x}{\sqrt{4k_1t}}\right) \tag{63}$$

$$u_2(x,t) = A_2 + B_2\mathrm{erf}\left(\frac{x}{\sqrt{4k_2t}}\right) \tag{64}$$

$$\xi(t) = \alpha\sqrt{t} \tag{65}$$

where

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-z^2}\, dz, \tag{66}$$

$$B_1 = -\frac{c_1}{\operatorname{erf}\left(\alpha/\sqrt{4k_1}\right)}, \quad B_2 = \frac{c_2}{1-\operatorname{erf}\left(\alpha/\sqrt{4k_2}\right)} \tag{67}$$

$$A_2 = -\operatorname{erf}\left(\alpha/\sqrt{4k_2}\right) B_2 \tag{68}$$

and α is a solution of the transcendental equation

$$\frac{\sqrt{k_1} c_1 e^{-\frac{\alpha^2}{4k_1}}}{\operatorname{erf}\left(\alpha/\sqrt{4k_1}\right)} + \frac{\sqrt{k_2} c_2 e^{-\frac{\alpha^2}{4k_2}}}{\left[1-\operatorname{erf}\left(\alpha/\sqrt{4k_2}\right)\right]} = -\frac{\gamma\sqrt{\pi}}{2}\alpha \tag{69}$$

The equation above has a unique solution $\alpha > 0$.

To write the problem as a one-phase Stefan problem, we simply set $c_2 = 0$, so $u_2 \equiv 0$, and the function $u = u_1$ satisfies

$$\frac{\partial u}{\partial t} - k_1 \frac{\partial^2 u}{\partial x^2} = 0,\ 0 < x < \xi(t),\ t > 0 \tag{70}$$

$$u(0,t) = c_1,\ t > 0 \tag{71}$$

$$u(x,0) = c_1,\ x < \xi(0) \tag{72}$$

$$u(\xi(t),t) = 0,\ t > 0 \tag{73}$$

$$k_1 \frac{\partial u}{\partial x}(\xi(t),t) = \gamma \frac{d\xi}{dt}(t) \tag{74}$$

where $k_1, k_2, \gamma > 0$ are given. The analytic solution is

$$u(x,t) = c_1 + B_1 \operatorname{erf}\left(\frac{x}{\sqrt{4k_1 t}}\right) \tag{75}$$

$$\xi(t) = \alpha\sqrt{t} \tag{76}$$

where

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-z^2}\, dz, \tag{77}$$

$$B_1 = -\frac{c_1}{\operatorname{erf}\left(\alpha/\sqrt{4k_1}\right)}, \tag{78}$$

and α is a solution of the transcendental equation

$$\frac{\sqrt{k_1} c_1 e^{-\frac{\alpha^2}{4k_1}}}{\operatorname{erf}\left(\alpha/\sqrt{4k_1}\right)} = -\frac{\gamma\sqrt{\pi}}{2}\alpha \tag{79}$$

We calculate

$$u_x(x,t) = \frac{B_1}{\sqrt{\pi k_1 t}} \exp\left(-\frac{x^2}{4k_1 t}\right) \tag{80}$$

so

$$k_1 u_x(s(t),t) = B_1 \frac{k_1}{\sqrt{\pi k_1 t}} \exp\left(-\frac{\alpha^2}{4k_1}\right) \tag{81}$$

$$= \frac{c_1 \sqrt{k_1} \exp\left(-\frac{\alpha^2}{4k_1}\right)}{\operatorname{erf}\left(\alpha/\sqrt{4k_1}\right)} \left(\frac{-1}{\sqrt{\pi t}}\right) \tag{82}$$

By virtue of α satisfying the above transcendental equation and $\frac{d\xi}{dt} \equiv \frac{\alpha}{2\sqrt{t}}$, it follows that

$$u_x(s(t),t) = \gamma \frac{\sqrt{\pi}}{2} \alpha \left(\frac{1}{\sqrt{\pi t}}\right) \tag{83}$$

$$= \gamma \frac{\alpha}{2\sqrt{t}} = \gamma \frac{d\xi}{dt} \tag{84}$$

That is, the Neumann boundary condition is satisfied at $x = s(t)$. Similarly,

$$k_1 u_x(0,t) = \frac{k_1 B_1}{\sqrt{\pi k_1 t}} = \frac{\sqrt{k_1} B_1}{\sqrt{\pi t}}$$

is the condition on the fixed boundary.

Note that the domain degenerates at $t = 0$ (and the problem data becomes singular), which is not supported by our theoretical framework; therefore, the data is shifted in time by a fixed amount `tShift`. The `fzero` rootfinder is used to solve the necessary transcendental equation to find α (denoted by `boundary_constant` in the code.) This solution is implemented in `true_solution.m`, included in Section A.6. The test code in `test_true_solution.m`, included in Section A.7, simply verified that the output shape and size of the parameters is correct.

Appendix A. Code Listings

sec:code-listing-forward

```
function [au_xx_S, u_x_S, u_S, u_T, u] = Forward(xmesh, tmesh, svals
    , avals, g, uInitial)
  % Forward: Solve forward problem on given mesh with given data.
  % Input Arguments:
  %     - xmesh: Space discretization on [0,1]
  %     - tmesh: Time discretization
  %     - svals: Position of boundary at time grid points
  %     - avals: Diffusion coefficient a(t) at time grid points
  %     - g: Function a(t) u_x(0, t) =: g(t)
  %     - uInitial: Function u(x,0)
  %
  % Output Arguments:
  %     - au_xx_S: Vector of trace values (a u_x)_x on x=s(t)
  %     - u_x_S: Vector of trace values u_x on x=s(t)
  %     - u_S: Vector of trace values u on x=s(t)
  %     - u_T: Vector of trace values u on t=T after transformation x
      /s(t)
  %    - u: Matrix of output values. Rows are time-levels, columns
     are space-levels

%%%
%%% Set up solver/parameters
%%%

% Define the vector of derivatives for s values
s_der = est_deriv(svals, tmesh);

% Interpolate s values to form s(t)
s_current = @(t) interp1(tmesh, svals, t);

% Interpolate a values to form a(t)
af = @(t) interp1(tmesh, avals, t);

% Interpolate s' values to form s'(t)
sder = @(t) interp1(tmesh, s_der, t);

%%%
%%% Calculate solution u(y,t) on rectangular domain
%%%
u = pdeSolver(xmesh, tmesh, af, s_current, sder, g, uInitial);

%%%
%%% Calculate values of solution at requested locations
%%%

%% Values of u(y,t) at t=T. Note that these values are not
   transformed to
% the non-rectangular domain.
u_T = u(end, :);

%% Values of u(y,t) at y=1, which is u(s(t), t).
u_S = u(:, end);

% spatial stepsize
h = xmesh(2) - xmesh(1);

% In order to compute u_x(x,t) from the output from pdeSolver, which
% corresponds to \tilde{u}(y,t)=psi(ys(t),t), we need to return
% psi_x(s(t),t) = \tilde{u}_y(1,t)/s(t) = u_x(:, end) ./ svals
u_x = est_x_partial(u, h);
```

```
u_x_S = u_x(:, end) ./ svals;

% Similarly, we compute a(t) u_{xx}(s(t), t) = a(t)\tilde{u}_{yy}(1,
    t)/s^2(t)
u_xx = est_x_partial(u_x, h);
au_xx_S = (avals .* u_xx(:, end)) ./ (svals.^2);

end
%%% End Main Function

%%% Begin Subfunctions
function u = pdeSolver(xmesh, tmesh, af, sf, sder, g, uTrue0)
    m = 0; % Symmetry of the problem. 0=slab (rectangular)

    % PDE to be solved is
    %    c u_t = x^{-m} (x^m f(x,t,u, DuDx))_x + s
    % The output values below give the corresponding value at each
        point (x, t, u, u_x)
    pde = @(x,t,u,DuDx) ...
           deal(...
                 sf(t).^2, ... # c
                 af(t)*DuDx, ... # f
                 sf(t)*sder(t)*x*DuDx ... # s
               );

    % Initial condition u(x,t_0)
    ic = uTrue0;

    % Boundary condition at x=xl and x=xr.
    % Suffix l corresponds to left-hand side of domain,
    % Suffix r corresponds to right-hand side of domain.
    % Boundary condition takes the form
    %    p(x,t,u) + q(x,t) f(x,t,u,u_x) = 0
    % where f is defined below.
    % In particular, the inputs correspond to (x,u) at x=xl and (x,u
       ) at x=xr
    % and the time t, and the outputs correspond to (p,q) at x=xl
        and (p,q) at
    % x=xr
    bc = @(xl, ul, xr, ur, t) ...
          deal(...
                g(t)*sf(t), ... # pl
                1, ... # ql
                -sder(t)*sf(t), ... # pr
                1 ... # qr
              );

    % Calculate solution evaluated at each point of xmesh and tmesh.
    sol = pdepe(m, pde, ic, bc, xmesh, tmesh);

    % Extract the first solution component as u.
    % Note: This solution is defined on tmesh * xmesh.
    % TODO: Transform to function u(x,t) within this routine.
    u = sol(:, :, 1);
end
%%% End Subfunctions
```

sec:code-listing-test-forward

```
function errorOut = test_Forward(len_xmesh, len_tmesh)
  % len_xmesh: Number of space grid points
  % len_tmesh: Number of time grid points

  xmesh = linspace(0,1,len_xmesh); % Space discretization for
     forward problem
  tmesh = linspace(0,1,len_tmesh)'; % Time discretization for
     forward problem

  [~, u_true_0, ~, g, ~, s_true, u_true, a_true] = true_solution(
     tmesh);

  boundary_values = s_true(tmesh);
  avals = a_true(tmesh);

  [au_xx_S, u_x_S, u_S, u_T, u] = ...
    Forward(xmesh, tmesh, boundary_values, avals, g, u_true_0);

  % Define grid for Forward problem solution
  [X, T] = meshgrid(xmesh, tmesh);

  % u:[0, max(s)] x [0, 1] \to R is defined by interpolation.
  u_interp = @(x,t) interp2(X, T, u, x/s_true(t), t, 'linear', NaN);

  x_new = linspace(0, max(boundary_values), len_xmesh);

  diff_visual = zeros(len_tmesh, len_xmesh);
  for i = 1:len_tmesh
```

```
    for j = 1:len_xmesh
      if x_new(j) > boundary_values(i)
        break
      end
      diff_visual(i, j) = u_true(x_new(j), tmesh(i)) - u_interp(
         x_new(j), tmesh(i));
    end
  end

  % Check size of outputs
  assert (all(size(u) == [len_tmesh, len_xmesh]));
  assert (length(au_xx_S) == len_tmesh);
  assert (iscolumn(au_xx_S));
  assert (length(u_S) == len_tmesh);
  assert (iscolumn(u_S));
  assert (length(u_x_S) == len_tmesh);
  assert (iscolumn(u_x_S));
  assert (length(u_T) == len_xmesh);
  assert (isrow(u_T));

  errorOut = norm(diff_visual, 'fro')*(x_new(2)-x_new(1))*(tmesh(2)-
     tmesh(1));
end
```

sec:code-listing-adjoint

```
function [psi_t_S, psi_x_S, psi_S, psi_T, psi] = Adjoint(xmesh,tmesh
    ,svals,avals,u_T,w_meas,s_der,u_S,mu_meas)
  % Adjoint: Compute values of adjoint for ISP
  % Input Arguments:
  %     - xmesh: Space discretization
  %     - tmesh: Time discretization
  %     - svals: Position of boundary at time grid points
  %     - avals: Diffusion coefficient a(t) at time grid points
  %     - u_T: Vector of values u(x,T;v) after transformation y=x/s(t
     )
  %     - w_meas: Function of x, representing measurement u(x,T)=:w(x
     )
  %     - s_der: Vector of values s'(t)
  %     - u_S: Vector of trace values u(s(t),t;v)
  %     - mu_meas: Function of t, representing measurement u(s(t),t)
     =:mu(t)

%%%
```

```
%%% Set up solver/parameters
%%%

  t_final = tmesh(end);

  % Interpolate s(t) values to form \bar{s}(t)=s(T-t)
  s_bar = @(t) interp1(tmesh, svals, t_final - t);

  % Interpolate s' values to form s'(T-t)
  sder = @(t) interp1(tmesh, s_der, t_final - t);

  % Interpolate a values to form \bar{a}(t) = a(T-t)
  af = @(t) interp1(tmesh, avals, t_final - t);

  % u_T is not transformed to a non-rectangular domain by Forward.m
  uT = @(y) interp1(xmesh, u_T, y);

  uS = @(t) interp1(tmesh, u_S, t_final - t);

  mu = @(t) mu_meas(t_final - t);

  % w_meas needs to be transformed before being passed to pdeSolver.
  w = @(y) w_meas(y*svals(end));

%%%
%%% Calculate solution on rectangular domain. These represent \tilde
    {\psi}(y,t)=psi(ys(t),t)
%%%
psi = pdeSolver(xmesh, tmesh, af, s_bar, sder, uT, w, uS, mu);

%%%
%%% Calculate values derived from psi
%%%

% spatial stepsize
h = xmesh(2) - xmesh(1);

% time stepsize
tau = tmesh(2) - tmesh(1);

% Values of psi(y,t) at y=1, which is psi(s(t),t).
psi_S = psi(:, end);
```

```
% Values of psi(x,t) at t=T. Note that these values are not
   transformed to
% the non-rectangular domain.
psi_T = psi(end, :);

% In order to compute psi_x(x,t) from the output from pdeSolver,
   which
% corresponds to \tilde{psi}(y,t)=psi(ys(t),t), we need to return
% psi_x(s(t),t) = \tilde{\psi}_y(1,t)/s(t) = psi_x(:, end) ./ svals
psi_x = est_x_partial(psi, h);
psi_x_S = psi_x(:, end) ./ svals;

% TODO: Compute these using second derivative information through
   PDE.
% This would allow Adjoint and Forward to use nearly the same post-
   processing
% step.
% In a similar way to above, we would like to produce values of
   psi_t(x,t),
% but pdeSolver produces values \tilde{psi}(y,t)=psi(ys(t),t)
% Hence we return
% psi_t(x,t) = - x s'(t) / s^2(t) \tilde{\psi}_y(y,t) + \tilde{\psi}
   _t(y,t)
%              = - y s'(t)/s(t) \tilde{\psi}_y(y,t) + \tilde{\psi}_t(y
   ,t)
% We first compute \tilde{psi}_t(y,t):
psi_t = est_t_partial(psi, tau);
% We need only the trace at x=s(t), which corresponds to y=1
psi_t_S = -s_der .* psi_x(:, end) + psi_t(:, end);

end
% End Main Function


% Begin Subfunctions
function psi = pdeSolver(xmesh, tmesh, af, sf, sder, uT, uTrueT, uS,
    mU)
  m = 0; % Symmetry of the problem. 0=slab (rectangular)

  % PDE to be solved is
  %    c u_t = x^{-m} (x^m f(x,t,u, DuDx))_x + s
  % The output values below give the corresponding value at each
     point (x, t, u, u_x)
  pde = @(x,t,u, DuDx) ...
         deal(...
          sf(t)^2, ... #c
          af(t)*DuDx, ... #f
          sf(t)*sder(t)*x*DuDx ... #s
         );

  % Initial condition u(x,t_0)
  ic = @(x) 2*(uT(x)-uTrueT(x));

  % Boundary condition at x=xl and x=xr.
  % Suffix l corresponds to left-hand side of domain,
  % Suffix r corresponds to right-hand side of domain.
  % Boundary condition takes the form
  %    p(x,t,u) + q(x,t) f(x,t,u,u_x) = 0
  % where f is defined below.
  % In particular, the inputs correspond to (x,u) at x=xl and (x,u)
     at x=xr
  % and the time t, and the outputs correspond to (p,q) at x=xl and
     (p,q) at
  % x=xr
  bc = @(xl, ul, xr, ur, t) ...
        deal(...
              0, ... # pl
              1, ... # ql
              sf(t)*(-sder(t)*ur+2*(uS(t)-mU(t))), ... # pr
              1 ... # qr
           );

  % Calculate solution evaluated at each point of xmesh and tmesh.
  sol = pdepe(m, pde, ic, bc, xmesh, tmesh);

  % Extract the first solution component as psi.
  % Note: This solution is defined on tmesh * xmesh.
  % Note: This last function reverses time so psi is "defined" on
     (0,1)x(0,T)
  psi = flipud(sol(:, :, 1));
end
% End Subfunctions
```

sec:code-listing-test-adjoint

```
function psiErrorOut = test_Adjoint(len_xmesh, len_tmesh,
   oscillation)
```

```
% test_Adjoint: Test for correct functioning of Adjoint function.
% Input Argument:
%     - len_xmesh: Number of space grid points
%     - len_tmesh: Number of time grid points
%     - oscillation: Amount to vary the data for Adjoint inputs.
%       The expectation for the adjoint problem is that as the
   boundary data vanishes, the adjoint vanishes.
if ~exist('oscillation', 'var')
  oscillation = 0;
end

xmesh = linspace(0,1,len_xmesh); % Space discretization for
   forward problem
tmesh = linspace(0,1,len_tmesh)'; % Time discretization for
   forward problem

[u_true_T, ~, u_true_S, ~, ~, s_true, ~, a_true] = true_solution(
   tmesh);

boundary_values = s_true(tmesh);
s_der = est_deriv(boundary_values, tmesh);

avals = a_true(tmesh);

% Set up problem with analytic solution as input for adjoint.
w_meas = u_true_T;
u_T = u_true_T(xmesh) + oscillation;

mu_meas = u_true_S;
u_S = mu_meas(tmesh) + oscillation;

% Run solver
[psi_t_S, psi_x_S, psi_S, psi_T, psi] = ...
  Adjoint(xmesh, tmesh, boundary_values, avals, u_T, w_meas, s_der
     , u_S, mu_meas);

% Check size of outputs
assert (all(size(psi) == [len_tmesh, len_xmesh]));
assert (length(psi_T) == len_xmesh);
assert (isrow(psi_T));
assert (length(psi_t_S) == len_tmesh);
assert (iscolumn(psi_t_S));
assert (length(psi_x_S) == len_tmesh);
```

```
  assert (iscolumn(psi_x_S));
  assert (length(psi_S) == len_tmesh);
  assert (iscolumn(psi_S));

  psiErrorOut = norm(psi, 'fro')*(xmesh(2)-xmesh(1))*(tmesh(2)-tmesh
      (1));
end
```

sec:code-listing-grad-s

```
function [grad]=grad_s(tmesh,s_der,svals,w_meas,u_T,mu_meas,u_x_S,
   psi_x_S,psi_t_S,psi_S,u_S,au_xx_S,s_star,psi_T)
  % grad_s: Calculate gradient with respect to x=s(t)
  % Input Arguments:
  %    - tmesh: Tiem grid on which functions are evaluated
  %    - s_der: Vector of derivative values for x=s(t)
  %    - svals: Vector of boundary location values
  %    - w_meas: Function, measurement u(x,T)
  %    - u_T: Vector of values representing u(x/s(T),T)
  %    - mu_meas: Function, measurement u(s(t),t)
  %    - u_x_S: Vector of values u_x(s(t),t)
  %    - psi_x_S: Vector of values psi_x(s(t),t)
  %    - psi_t_S: Vector of values psi_t(s(t),t)
  %    - psi_S: Vector of values psi(s(t),t)
  %    - u_S: Vector of values u(s(t),t)
  %    - au_xx_S: Vector of values a(t) u_{xx}(s(t),t)
  %    - s_star: Measurement s(T)
  %    - psi_T: Vector of values psi(x/s(T),T)
  % Output arguments:
  %    - grad: Vector of values representing J_s(t)

  t_final = tmesh(end);

  %% Test code for grad generation
  % grad=rand(length(tmesh),1);

  %% This code should mirror the gradient formula for the control x=
      s(t)
  % J_s(t) = [2 (u-mu) u_x - psi a u_xx + psi_x s' + psi_t]_{x=s(t)}
  % J_s(T) = |u(s(T),T)-w(s(T))|^2 + 2 (s(T)-s_*) - psi(s(T),T)
  % Note in particular that all of the vectors here should be time-
      like (column vectors)
  grad = (...
           2*(u_S - mu_meas(tmesh)) .* u_x_S + ...
```

```
              psi_x_S .* s_der + ...
              psi_t_S - ...
              psi_S .* au_xx_S ...
            );
  grad(end) = (...
                 (w_meas(t_final)-u_T(end))^2 + ...
                 2*(svals(end)-s_star) - ...
                 psi_T(end) ...
               );
end
```

sec:code-listing-true-solution

```
function [u_true_T, u_true_0, u_true_S, aux_true_0, s_star, s_true,
    u_true, a_true] = true_solution(tmesh, k1, c1, latentHeat, tShift
    )
  % true_solution: Return parameter functions for "true" solution.
  % See notes.tex for a citation to Tikhonov & Samarskii
  % Input Argument:
  %     - tmesh: Grid of time values, or a vector [t_0, t_final]
  %     - k1, c1, latentHeat: Scalars representing constants in the
      example. Default: [1, -1, 1]
  %     - tShift: Time shift applied to model problem to avoid
  %       singular data.
  %
  % Output Arguments:
  %     - u_true_T: function u(x, t_final)
  %     - u_true_0: function u(x, t_0)
  %     - u_true_S: function u(s(t), t)
  %     - aux_true_0: function g(t) = a(t) u_x(0, t)
  %     - s_star: s(t_final)
  %     - s_true: function x=s(t)
  %     - u_true: function u=u(x,t)
  %     - a_true: function a=a(t)

  if ~exist('k1', 'var')
    k1 = 1;
  end
  if ~exist('c1', 'var')
    c1 = -1;
  end
  if ~exist('latentHeat', 'var')
    latentHeat = 1;
  end
```

```
if ~exist('tShift', 'var')
  tShift = 1e-1;
end

persistent boundary_constant
persistent B1

t_initial = tmesh(1);

t_final = tmesh(end);

% Analytic diffusion coefficient a(t)
a_true = @(t) k1*ones(size(t));

% Equation defining boundary_constant
if isempty(boundary_constant)
  boundary_constant_f = @(z) (z*latentHeat * sqrt(pi))/2 + (sqrt(
     k1)*c1*exp(-(z.^2) / (4*k1))) / erf(z / (2 * sqrt(k1)));
  % Assume that the root is in the interval (1e-3, 5)
  boundary_constant = fzero(boundary_constant_f, [1e-3, 5]);
end

% Once boundary_constant is known, B1 is fixed.
if isempty(B1)
  B1 = -c1/erf(boundary_constant/(2*sqrt(k1)));
end

% true s(t)
s_true = @(t) boundary_constant * sqrt(t+tShift);

% s(T) - s(T) at final time
s_star = s_true(t_final);

% Analytic function u=u(x,t)
u_true = @(x,t) c1 + B1*erf(x./(2*sqrt(k1*(t + tShift))));

% Analytic function u_x(x,t)
ux_true = @(x,t) (B1/sqrt(pi*k1*(t + tShift))).*exp(-(x.^2)./(4*k1
   *(t+tShift)));

% Analytic function g(t) = a u_x(0,t).
% This would easily vectorize by writing ux_true(0,t) as a special
    case
  aux_true_0_ptwise = @(t) a_true(t) .* ux_true(0, t); % == B1*sqrt(
     k1/(pi*(t + tShift)))
  aux_true_0 = @(t) arrayfun(aux_true_0_ptwise, t);

  % \mu(t) = u(s(t),t) Phase Transition temperature
  u_true_S = @(t) zeros(size(t)); % == u_true(s_true(t), t);

  % w(x) = u(x, T) measurement at final moment
  u_true_T_ptwise = @(x) u_true(x, t_final);
  u_true_T = @(x) arrayfun(u_true_T_ptwise, x);

  % phi(x) = u(x, 0) measurement at initial moment
  u_true_0_ptwise = @(x) u_true(x, t_initial);
  u_true_0 = @(x) arrayfun(u_true_0_ptwise, x);
end
```

sec:code-listing-test-true-solution

```
function [tf] = test_true_solution(len_xmesh, len_tmesh, visualize)
if ~exist('len_xmesh', 'var')
    len_xmesh = 100;
end
if ~exist('len_tmesh', 'var')
    len_tmesh = 100;
end
if ~exist('visualize', 'var')
    visualize = false;
end

tmesh = linspace(0,1,len_tmesh)'; % Time discretization for forward
   problem

[ ...
  u_true_T, u_true_0, u_true_S, ...
  aux_true_S, ...
  s_star, ...
  s_true, u_true, ...
  a_true ...
  ] = true_solution(tmesh);

boundary_values = s_true(tmesh);
xmesh = linspace(0, max(boundary_values), len_xmesh);

u_true_fullDomain = @(x,t) u_true(x,t).*(x<s_true(t));
```

```
% Define grid for Forward problem solution
[X, T] = meshgrid(xmesh, tmesh);

% Easiest to index U, X, and T with a common linear index (
   u_true_fullDomain
% is not currently vectorized)
U = zeros(size(X));
for i = 1:numel(X)
  U(i) = u_true_fullDomain(X(i), T(i));
end

% Check for correct shape and size of output arguments (without more
    care
% this _can_ go wrong!
assert(all(U(:) <= 0))
assert(isrow(u_true_T(xmesh)) && numel(u_true_T(xmesh)) == len_xmesh
   )
assert(isrow(u_true_0(xmesh)) && numel(u_true_0(xmesh)) == len_xmesh
   )
assert(iscolumn(u_true_S(tmesh)) && numel(u_true_S(tmesh)) ==
   len_tmesh)
assert(iscolumn(aux_true_S(tmesh)) && numel(aux_true_S(tmesh)) ==
   len_tmesh)
assert(iscolumn(s_true(tmesh)) && numel(s_true(tmesh)) == len_tmesh)
assert(iscolumn(a_true(tmesh)) && numel(a_true(tmesh)) == len_tmesh)
assert(isscalar(s_star) && s_star > 0)

if visualize
  % For graphing, remove values if they are close to zero or
     positive
  U(U>-1e-8) = NaN;

  figure();

  subplot(1,2,1);
  surf(X, T, U, 'EdgeColor', 'none');
  xlabel('x');
  ylabel('t');
  zlabel('u');
  title('Surface plot of Analytical Solution');

  subplot(1,2,2);
```

```
    hold on
    imagesc('XData', xmesh, 'YData', tmesh, 'CData', U);
    plot(s_true(tmesh), tmesh, 'color', 'red');
    xlabel('Distance x');
    ylabel('Time t');
    title('Analytical Solution and Boundary Curve');
    colorbar
    axis tight
    hold off

    figure()
    plot(tmesh, u_true_S(tmesh))
end

tf = true;
end
```

Appendix B. Change of Variables Details

The order with which variables are referenced has a meaning inside the adjoint and forward codes. In particular, the change of variables being taken, and when it is taken, affects the output, in general; in this small note we document the change of variables used to transform those problems to a rectangular domain, since it is implemented multiple times in the code and has led to issues before. It is essential for the convergence of the adjoint code that the following transformation be taken *before* reversing the problem in time. Define the change of variables

$$(x,t) \mapsto (y,\bar{t}),$$
$$\bar{t} = t, \quad y = x/s(t)$$

Define a new function

$$\tilde{\psi}(y,\bar{t}) = \psi(ys(\bar{t}),\bar{t})$$

If we have computed values of ψ in Ω, we may compute

$$\tilde{\psi}_y(y,\bar{t}) = \frac{\partial}{\partial y}\psi(ys(\bar{t}),\bar{t}) = \psi_x(ys(\bar{t}),\bar{t})s(\bar{t})$$
$$\tilde{\psi}_{yy}(y,\bar{t}) = \psi_{xx}(ys(\bar{t}),\bar{t})$$
$$\tilde{\psi}_{\bar{t}} = \frac{\partial}{\partial \bar{t}}\psi(ys(\bar{t}),\bar{t}) = \psi_x(ys(\bar{t}),\bar{t})s'(\bar{t}) + \psi_t(ys(\bar{t}),\bar{t})$$

The function ψ is defined by the inverse transformation,

$$(y,\bar{t}) \mapsto (x,t),$$
$$t = \bar{t}, \quad x = ys(\bar{t})$$

so

$$\psi(x,t) = \tilde{\psi}\left(x/s(t), t\right)$$

Suppose we have computed values $\tilde{\psi}(y, \bar{t})$ in the domain $[0,1] \times [0,T]$ and would like to compute those of ψ; we have

$$\psi_x(x,t) = \frac{\partial}{\partial x}\tilde{\psi}\left(x/s(t), t\right) = \tilde{\psi}_y\left(x/s(t), t\right)/s(t)$$

so in particular

$$\psi_x(s(t),t) = \tilde{\psi}_y(1,t)/s(t)$$
$$\psi_{xx}(x,t) = \tilde{\psi}_{yy}\left(x/s(t),t\right)/s^2(t)$$
$$\psi_t(x,t) = -xs'(t)/s^2(t)\tilde{\psi}_y\left(x/s(t),t\right) + \tilde{\psi}_{\bar{t}}(x/s(t),t)$$

Appendix C. Model Problem Zoo

C.1. **Separable Solution**. The model problem currently implemented in `test_Forward.m`, included in Section A.2 is (21)–(24) with a separable analytic solution and identically zero heat sources. That is, consider

$$Lu \equiv a(t)u_{xx} - u_t = f, \text{ in } \Omega$$
$$u(x,0) = \phi(x),\ 0 \le x \le s(0) = s_0$$
$$a(t)u_x(0,t) = g(t),\ 0 \le t \le T$$
$$a(t)u_x(s(t),t) + s'(t) = 0,\ 0 \le t \le T$$

where the analytic solution is given by

$$a(t) = a_{\text{true}}(t) \equiv 1, \tag{85}$$
$$u(x,t) = T(t)X(x) \tag{86}$$

so

$$u_x(x,t) = T(t)X'(x) \tag{87}$$

and

$$u_{xx}(x,t) = T(t)X''(x) \tag{88}$$
$$u_t = T'(t)X(x) \tag{89}$$

Choosing $T(0) = 1$ and asserting further that $T' = T$, we see that to satisfy (21) with right-hand side identically zero, we must have

$$u_{xx} - u_t = TX'' - T'X = T(X'' - X) = 0$$

Here, we see that we may take any solution of $X'' - X = 0$ satisfying a given boundary condition at $x = 0$, with price that the ODE that $s(t)$ must solve may be complicated. For instance, we may take the condition $u_x(0,t) = 0$, which implies

$$X(x) = \cosh(x)$$

is a solution, and hence

$$\phi(x) = \cosh(x), \tag{90}$$

$$u(x,t) = \exp(t)\cosh(x) \tag{91}$$

and the analytic function $g(t)$ is given by

$$g(t) = a(t)u_x(0,t) = \exp(t)\sinh(0) = 0 \tag{92}$$

To find the function $x = s(t)$, we solve the ODE (24), which takes the form

$$0 = T(t)X'(s(t)) + s'(t) = \exp(t)\sinh(s(t)) + s'(t), \tag{93}$$

Choosing the normalization

$$s(0) = 1 =: s_0 \tag{94}$$

we can check that the analytic solution is

$$s(t) \equiv 2\,\mathrm{arccoth}\left(\exp(\exp(t) - 1)\coth(1/2)\right) \tag{95}$$

Department of Mathematical Sciences, Florida Institute of Technology, Melbourne, FL 32901