Case Study # 1: 1D Transient Heat Diffusion

John Karasinski
Graduate Student Researcher
Center for Human/Robotics/Vehicle Integration and Performance
Department of Mechanical and Aerospace Engineering
University of California
Davis, California 95616
Email: karasinski@ucdavis.edu

1 Problem Description

The problem of 1D unsteady heat diffusion in a slab of unit length with a zero initial temperature and both ends maintained at a unit temperature can be described by:

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} \text{ subject to } \begin{cases} T(x,0^-) = 0 & \text{for } 0 \le x \le 1 \\ T(0,t) = T(1,t) = 1 & \text{for } t > 0 \end{cases} \tag{1}$$

and has the well known analytical solution:

$$T^*(x,t) = 1 - \sum_{k=1}^{\infty} \frac{4}{(2k-1)\pi} \sin[(2k-1)\pi x] \star \exp[-(2k-1)^2\pi^2 t]. \tag{2}$$

In addition to the analytical solution, several numerical methods can be employed to solve the diffusion equation. Two of these methods are derived in the following section.

2 Solution Algorithms

The Taylor-series (TS) method can be used on this equation to derive a finite difference approximation to the PDE. Applying the definition of the derivative,

$$f'(x) \approx \frac{f(x+\varepsilon) - f(x)}{\varepsilon} \tag{3}$$

to Eqn. (1) yields

$$\begin{aligned} \frac{\partial T}{\partial t} &= \frac{T(x,t+\Delta t) - T(x,t)}{\Delta t} \\ &= \frac{T_i^{k+1} - T_i^k}{\Delta t}. \end{aligned} \tag{4}$$

From the definition of the Taylor series,

$$f(x+\varepsilon) = f(x) + \varepsilon f'(x) + \frac{\varepsilon^2}{2} f''(x) + \dots \tag{5}$$

which, when applied to T_i^{k+1} and T_i^k gives

$$T_{i+1} = T_i + \Delta x \frac{\partial T_i}{\partial x} + \frac{\Delta x^2}{2} \frac{\partial^2 T_i}{\partial x^2} + O(\Delta x^3) \tag{6}$$

and

$$T_{i-1} = T_i - \Delta x \frac{\partial T_i}{\partial x} + \frac{\Delta x^2}{2} \frac{\partial^2 T_i}{\partial x^2} - O(\Delta x^3). \tag{7}$$

Adding Eqn. (6) and Eqn. (7) yields

$$T_{i+1} + T_{i-1} = 2T_i + \Delta x^2 \frac{\partial^2 T_i}{\partial x^2} + O(\Delta x^4) \tag{8}$$

which can be rearranged as the approximation for the second order term from Eqn. (1),

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1} + 2T_i + T_{i-1}}{\Delta x^2} + O(\Delta x^4), \tag{9}$$

and can also be combined with the the above equations to form

$$\frac{T_i^{k+1} - T_i^k}{\Delta t} \approx \frac{T_{i+1}^k - 2T_i^k + T_{i-1}^k}{\Delta x^2}. \tag{10}$$

This result can be arranged to form both Forward-Time, Centered-Space (FTCS) explicit and implicit schemes.

2.1 Explicit

Eqn. (10) can be rearranged to form the explicit scheme, which is

$$T_i^{k+1} = sT_{i+1}^k + (1-2s)T_i^k + sT_i^k \tag{11}$$

where

$$s = \frac{\alpha \Delta t}{\Delta x^2} \tag{12}$$

and α is the thermal diffusivity of the material.

This scheme can be implemented to solve the problem computationally. In pseudocode, looks like

```
while t ≤ t_end do
    i ← 1
    for i in N − 1 do
        T_{k+1}[i] = sT_k[i+1] + (1 − 2s)T_k[i] + sT_k[i−1]
        i ← i + 1
    end for
    T_k = T_{k+1}
    t ← t + dt
end while
```

where N is the number of elements in your mesh. Each element in the interior is looped over (the boundary conditions remain constant), and the time marches forward until the designated end time has been reached.

2.2 Implicit

Eqn. (10) can also be rearranged to form the implicit scheme, which is

$$T_i^k = -sT_{i+1}^{k+1} + (1+2s)T_i^{k+1} - sT_{i-1}^k \tag{13}$$

where again,

$$s = \frac{\alpha \Delta t}{\Delta x^2} \tag{14}$$

and α is the thermal diffusivity of the material.

A tridiagonal system for n unknowns may be written as $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$, where $a_1 = 0$ and $c_n = 0$.

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix} \tag{15}$$

The forward sweep consists of modifying the coefficients as follows, denoting the new modified coefficients with primes:

$$c'_i = \begin{cases} \dfrac{c_i}{b_i} & ; i = 1 \\ \dfrac{c_i}{b_i - a_i c'_{i-1}} & ; i = 2, 3, \dots, n-1 \end{cases} \tag{16}$$

and

$$d'_i = \begin{cases} \dfrac{d_i}{b_i} & ; i = 1 \\ \dfrac{d_i - a_i d'_{i-1}}{b_i - a_i c'_{i-1}} & ; i = 2, 3, \dots, n. \end{cases} \tag{17}$$

The solution is then obtained by back substitution:

$$\begin{aligned} x_n &= d'_n \\ x_i &= d'_i - c'_i x_{i+1} \qquad ; i = n-1, n-2, \dots, 1. \end{aligned} \tag{18}$$

3 Results

A Python script was used to obtain results for a 21 point mesh (N=21), and the Root Mean Square error,

$$\text{RMS} = \frac{1}{N^2} \sqrt{\sum_{i=1}^{N} [T_i^n - T^*(x_i, t_n)]^2} \tag{19}$$

was obtained for $s(= \Delta t / \Delta x^2)$ = 1/6, 0.25, 0.5, and 0.75, at t = 0.03, 0.06, and 0.09 using both the explicit and the implicit methods.

t	Explicit RMS	Implicit RMS
0.03	4.41E-4	3.46E-5
0.06	5.66E-4	3.33E-6
0.09	6.26E-4	9.44E-6

Table 1. RMS results from the numerical simulations compared to the analytic solution for s = 1/6

t	Explicit RMS	Implicit RMS
0.03	6.59E-4	2.24E-5
0.06	9.05E-4	6.06E-6
0.09	9.61E-4	9.35E-6

Table 2. RMS results from the numerical simulations compared to the analytic solution for s = 0.25

The RMS between the implicit and analytic solutions and the explicit and analytic solutions are shown in Table 3. The RMS tended to grow as a function of s, and shrink as a function of t. Additionally, the RMS for the explicit solution tended to be two orders of magnitude larger than the RMS for the implicit solution.

t	Explicit RMS	Implicit RMS
0.03	1.74E-3	3.77E-5
0.06	2.24E-3	1.53E-5
0.09	2.21E-3	1.95E-5

Table 3. RMS results from the numerical simulations compared to the analytic solution for s = 0.5

t	Explicit RMS	Implicit RMS
0.03	3.87E-3	5.38E-5
0.06	4.21E-3	2.46E-5
0.09	3.51E-3	2.96E-5

Table 4. RMS results from the numerical simulations compared to the analytic solution for s = 0.75

4 Discussions

Shankle chicken tail, fatback short ribs meatball pancetta ball tip sirloin short loin. Pork tongue pork belly pork loin beef ribs. Shank turkey pork belly pork loin ham hock ball tip leberkas meatloaf chuck ground round filet mignon kielbasa sirloin turducken tri-tip. Pancetta brisket sirloin beef ribs spare ribs, swine bacon ham hock. Ham kielbasa corned beef turkey turducken. Kevin biltong pork, tenderloin chuck pig ball tip filet mignon.

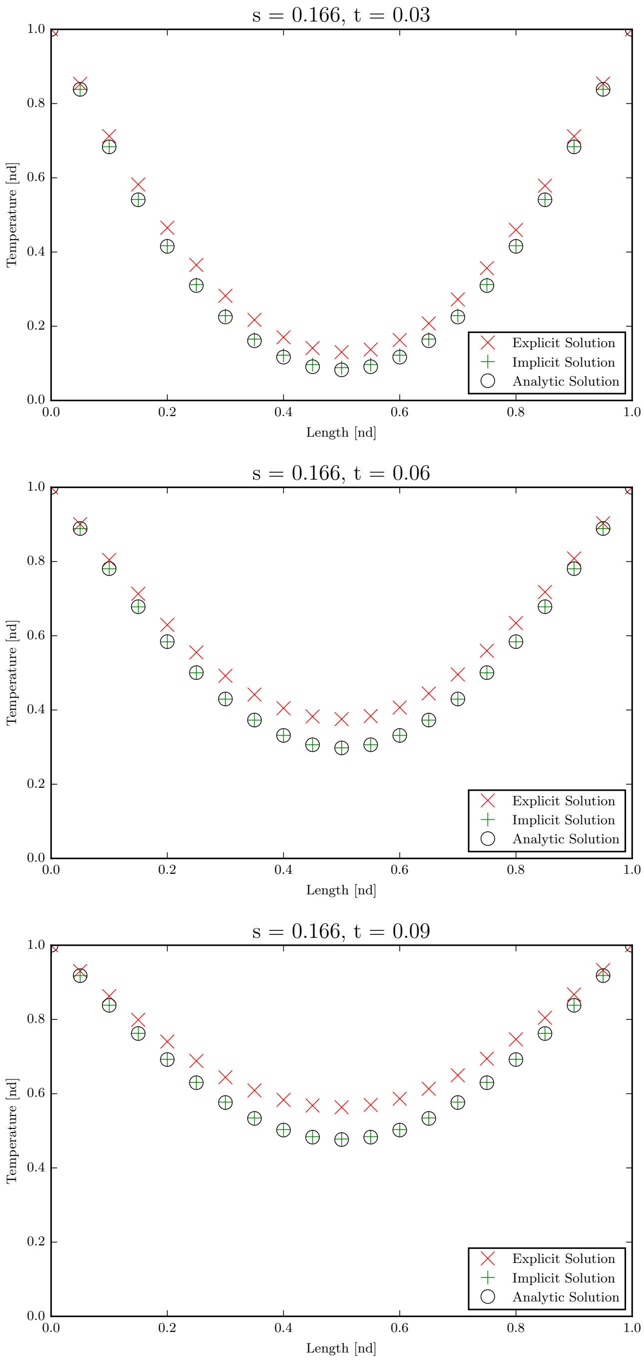

Fig. 1. Results for $s = 1/6$

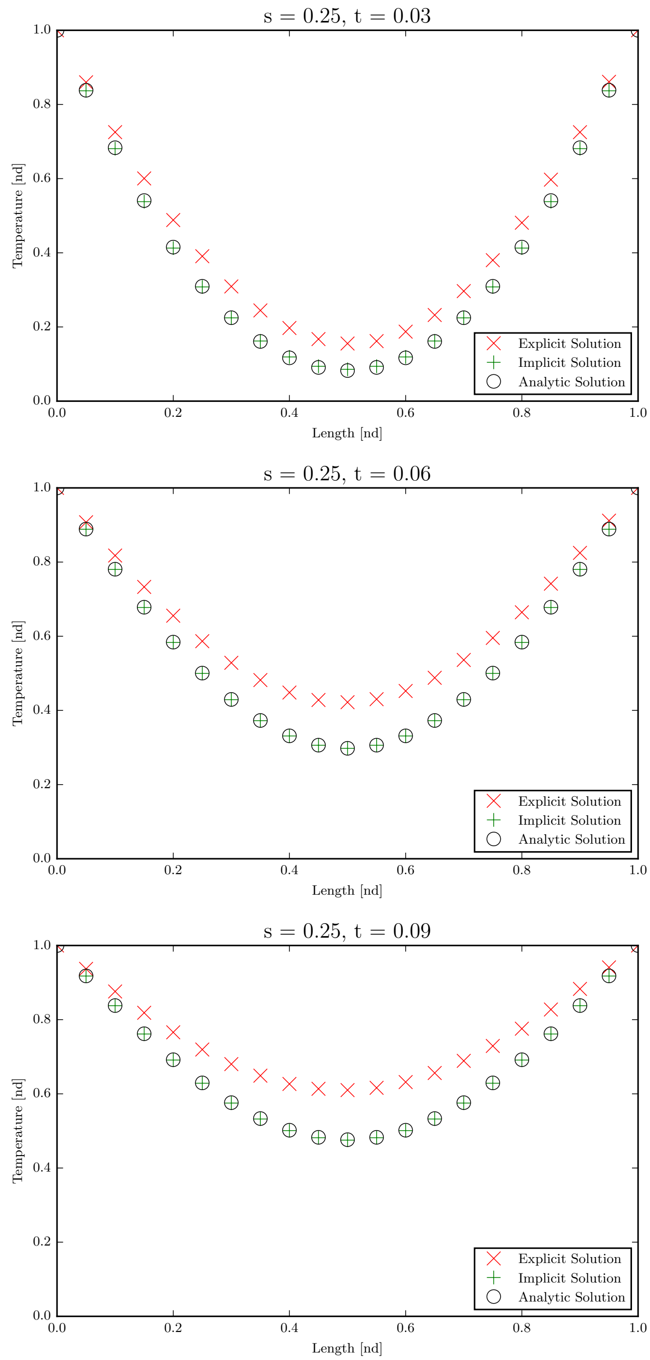

Fig. 2. Results for $s = 0.25$

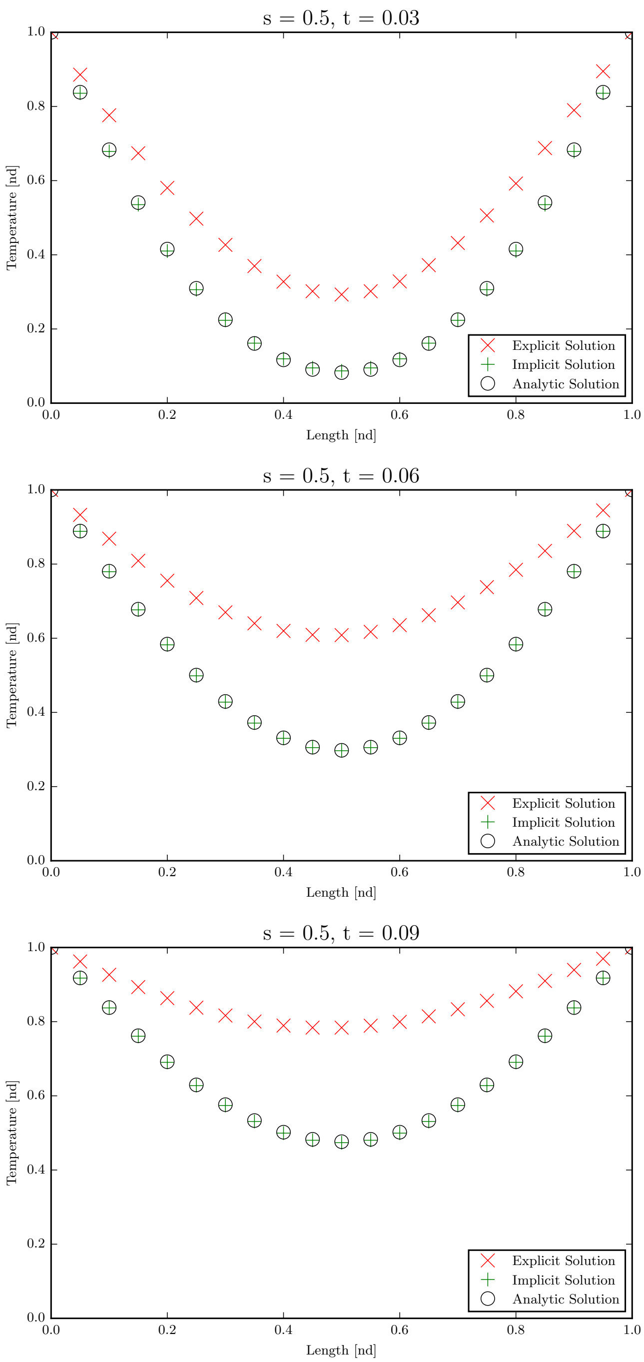

Fig. 3. Results for $s = 0.5$

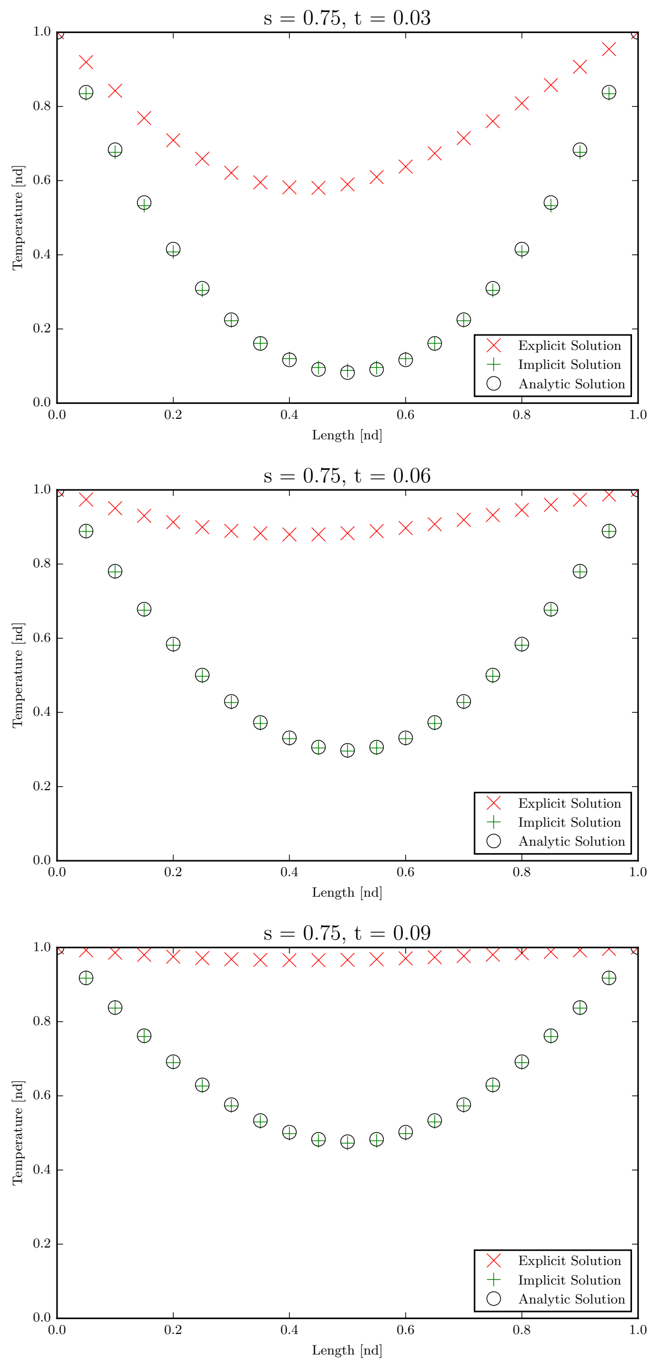

Fig. 4. Results for $s = 0.75$

Appendix A: Python Code

```python
import numpy as np
import matplotlib.pyplot as plt
import os

# Configure figures for production
WIDTH = 495.0  # the number latex spits out
FACTOR = 1.0   # the fraction of the width you'd like the figure to occupy
fig_width_pt  = WIDTH * FACTOR

inches_per_pt = 1.0 / 72.27
golden_ratio  = (np.sqrt(5) - 1.0) / 2.0  # because it looks good

fig_width_in  = fig_width_pt * inches_per_pt  # figure width in inches
fig_height_in = fig_width_in * golden_ratio   # figure height in inches
fig_dims      = [fig_width_in, fig_height_in] # fig dims as a list


def Solver(s, t_end, show_plot=False):
    # Problem Parameters
    L = 1.            # Domain lenghth       [n.d.]
    T0 = 0.           # Initial temperature  [n.d.]
    T1 = 1.           # Boundary temperature [n.d.]
    N = 21

    # Set-up Mesh
    x = np.linspace(0, L, N)
    dx = x[1] - x[0]

    # Calculate time-step
    dt = s * dx ** 2.0

    # Initial Condition with boundary conditions
    T_initial = [T0] * N
    T_initial[0] = T1
    T_initial[N - 1] = T1

    # Explicit Numerical Solution
    T_explicit = Explicit(np.array(T_initial).copy(), t_end, dt, s)

    # Implicit Numerical Solution
    T_implicit = Implicit(np.array(T_initial).copy(), t_end, dt, s)

    # Analytical Solution
    T_analytic = np.array(T_initial).copy()
    for i in range(0, N):
        T_analytic[i] = Analytic(x[i], t_end)

    # Find the RMS
    RMS = RootMeanSquare(T_implicit, T_analytic)
    ExplicitRMS = RootMeanSquare(T_explicit, T_analytic)

    # Format our plots
    plt.figure(figsize=fig_dims)
    plt.axis([0, L, T0, T1])
    plt.xlabel('Length [nd]')
    plt.ylabel('Temperature [nd]')
    plt.title('s = ' + str(s)[:5] + ', t = ' + str(t_end)[:4])

    # ...and finally plot
    plt.plot(x, T_explicit, 'xr', markersize=9, label='Explicit Solution')
    plt.plot(x, T_implicit, '+g', markersize=9, label='Implicit Solution')
    plt.plot(x, T_analytic, 'ob', markersize=9, mfc='none', label='Analytic Solution')
    plt.legend(loc='lower right')

    # Save plots
    save_name = 'proj_1_s_' + str(s)[:5] + '_t_' + str(t_end) + '.pdf'
```

```python
    try:
        os.mkdir('figures')
    except Exception:
        pass

    plt.savefig('figures/' + save_name, bbox_inches='tight')
    if show_plot:
        plt.show()
    plt.clf()

    return RMS, ExplicitRMS


def Explicit(Told, t_end, dt, s):
    """
    This function computes the Forward-Time, Centered-Space (FTCS) explicit
    scheme for the 1D unsteady heat diffusion problem.
    """
    N = len(Told)
    time = 0.
    Tnew = Told

    while time <= t_end:
        for i in range(1, N - 1):
            Tnew[i] = s * Told[i + 1] + (1 - 2.0 * s) * Told[i] + s * Told[i - 1]

        Told = Tnew
        time += dt

    return Told


def Implicit(Told, t_end, dt, s):
    """
    This function computes the Forward-Time, Centered-Space (FTCS) implicit
    scheme for the 1D unsteady heat diffusion problem.
    """
    N = len(Told)
    time = 0.

    # Build our 'A' matrix
    a = [-s] * N
    a[0], a[-1] = 0, 0
    b = [1 + 2 * s] * N
    b[0], b[-1] = 1, 1        # hold boundary
    c = a

    while time <= t_end:
        Tnew = TDMAsolver(a, b, c, Told)

        Told = Tnew
        time += dt

    return Told


def RootMeanSquare(a, b):
    """
    This function will return the RMS between two lists (but does no checking
    to confirm that the lists are the same length).
    """
    N = len(a)

    RMS = 0.
    for i in range(0, N):
        RMS += (a[i] - b[i]) ** 2.

```

```python
    RMS = RMS ** (1. / 2.)
    RMS /= N**2.

    return RMS


def TDMAsolver(a, b, c, d):
    """
    Tridiagonal Matrix Algorithm (a.k.a Thomas algorithm).
    """
    N = len(a)
    Tnew = d

    # Initialize arrays
    gamma = np.zeros(N)
    xi = np.zeros(N)

    # Step 1
    gamma[0] = c[0] / b[0]
    xi[0] = d[0] / b[0]

    for i in range(1, N):
        gamma[i] = c[i] / (b[i] - a[i] * gamma[i - 1])
        xi[i] = (d[i] - a[i] * xi[i - 1]) / (b[i] - a[i] * gamma[i - 1])

    # Step 2
    Tnew[N - 1] = xi[N - 1]

    for i in range(N - 2, -1, -1):
        Tnew[i] = xi[i] - gamma[i] * Tnew[i + 1]

    return Tnew


def Analytic(x, t):
    """
    The analytic answer is 1 - Sum(terms). Though there are an infinite
    number of terms, only the first few matter when we compute the answer.
    """
    result = 1
    large_number = 1E6

    for k in range(1, int(large_number) + 1):
        term = ((4. / ((2. * k - 1.) * np.pi)) *
                np.sin((2. * k - 1.) * np.pi * x) *
                np.exp(-(2. * k - 1.) ** 2. * np.pi ** 2. * t))

        # If subtracting the term from the result doesn't change the result
        # then we've hit the computational limit, else we continue.
        # print '{0} {1}, {2:.15f}'.format(k, term, result)
        if result - term == result:
            return result
        else:
            result -= term


def main():
    """
    Main function to call solver over assigned values and create some plots to
    look at the trends in RMS compared to s and t.
    """
    # Loop over requested values for s and t
    s = [1. / 6., .25, .5, .75]
    t = [0.03, 0.06, 0.09]

    RMS = []
    with open('results.dat', 'w+') as f:
```

```
        for i, s_ in enumerate(s):
            sRMS = [0] * len(t)
            for j, t_ in enumerate(t):
                sRMS[j], ExplicitRMS = Solver(s_, t_, False)
                f.write('{0:.3f} {1:.2f} {2:.2e} {3:.2e} \n'.format(s_, t_, sRMS[j], ExplicitRMS))
                # print i, j, sRMS[j]
            RMS.append(sRMS)

    # Convert to np array to make this easier...
    RMS = np.array(RMS)

    # Check for trends in RMS vs t
    plt.figure(figsize=fig_dims)
    plt.plot(t, RMS[0], '.r', label='s = 1/6')
    plt.plot(t, RMS[1], '.g', label='s = .25')
    plt.plot(t, RMS[2], '.b', label='s = .50')
    plt.plot(t, RMS[3], '.k', label='s = .75')
    plt.xlabel('t')
    plt.ylabel('RMS')
    plt.title('RMS vs t')
    plt.legend(loc='best')

    save_name = 'proj_1_rms_vs_t.pdf'
    plt.savefig('figures/' + save_name, bbox_inches='tight')
    plt.clf()

    # Check for trends in RMS vs s
    plt.figure(figsize=fig_dims)
    plt.plot(s, RMS[:, 0], '.r', label='t = 0.03')
    plt.plot(s, RMS[:, 1], '.g', label='t = 0.06')
    plt.plot(s, RMS[:, 2], '.b', label='t = 0.09')
    plt.xlabel('s')
    plt.ylabel('RMS')
    plt.title('RMS vs s')
    plt.legend(loc='best')

    save_name = 'proj_1_rms_vs_s.pdf'
    plt.savefig('figures/' + save_name, bbox_inches='tight')
    plt.clf()

if __name__ == "__main__":
    main()
```