

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:

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

and has the well known analytical solution:

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

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,

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon} \quad (3)$$

to Eqn. (1) yields:

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

From the definition of the Taylor series,

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

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

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

and

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

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

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

we can also combine the approximation for the second order term from Eqn. (1) to find

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

which we can combine with the above equations to form

$$\frac{T_i^{k+1} - T_i^k}{\Delta t} \approx \frac{T_{i+1}^k - 2T_i^k + T_{i-1}^k}{\Delta x^2} \quad (10)$$

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

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

and the implicit scheme is

$$T_i^k = -sT_{i+1}^{k+1} + (1+2s)T_i^{k+1} - sT_{i-1}^{k+1} \quad (12)$$

where

$$s = \frac{\alpha \Delta t}{\Delta x^2} \quad (13)$$

and α is the thermal diffusivity of the material.

t	Implicit RMS	Explicit RMS
0.03	7.27E-4	9.25E-3
0.06	6.99E-5	1.19E-2
0.09	1.98E-4	1.31E-2

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

t	Implicit RMS	Explicit RMS
0.03	4.70E-4	1.38E-2
0.06	1.27E-4	1.90E-2
0.09	1.96E-4	2.02E-2

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

t	Implicit RMS	Explicit RMS
0.03	7.92E-4	3.66E-2
0.06	3.20E-4	4.71E-2
0.09	4.10E-4	4.63E-2

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

t	Implicit RMS	Explicit RMS
0.03	1.13E-3	8.13E-2
0.06	5.17E-4	8.83E-2
0.09	6.22E-4	7.38E-2

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.

3 Results

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

$$\text{RMS} = \frac{1}{N} \sqrt{\sum_{i=1}^N [T_i^n - T^*(x_i, t_n)]^2} \quad (14)$$

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 .

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.

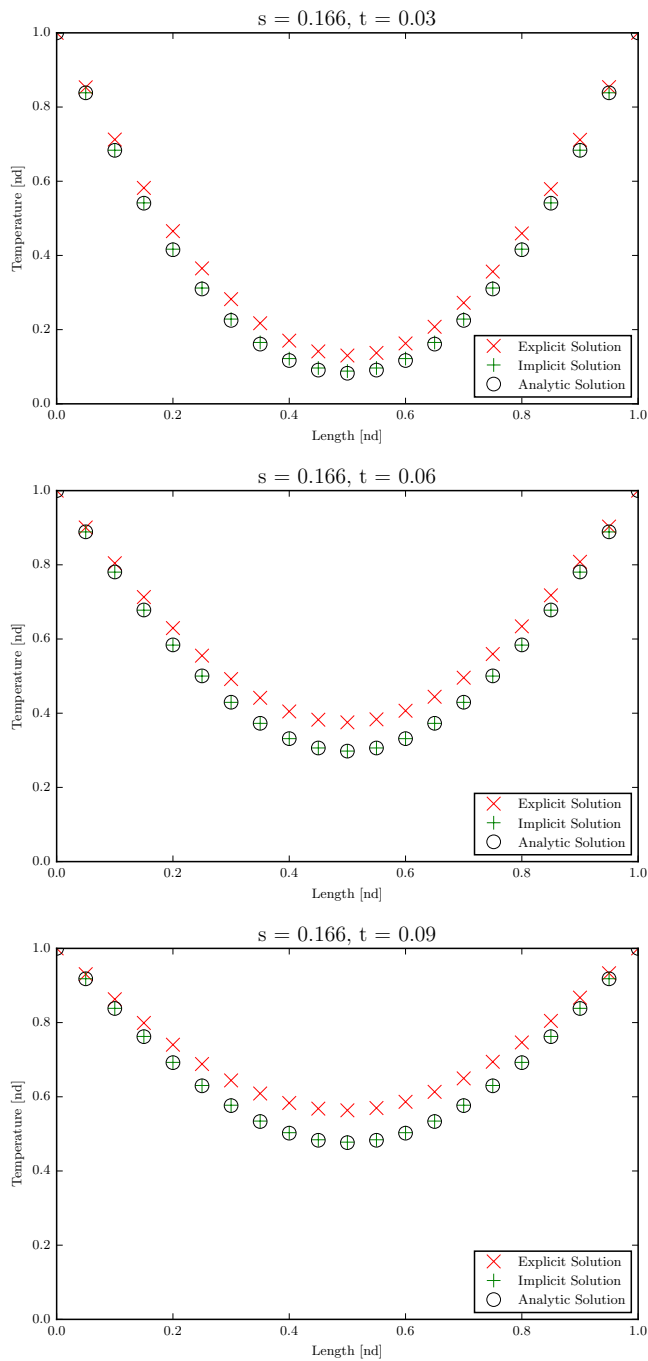


Fig. 1. Results for $s = 1/6$

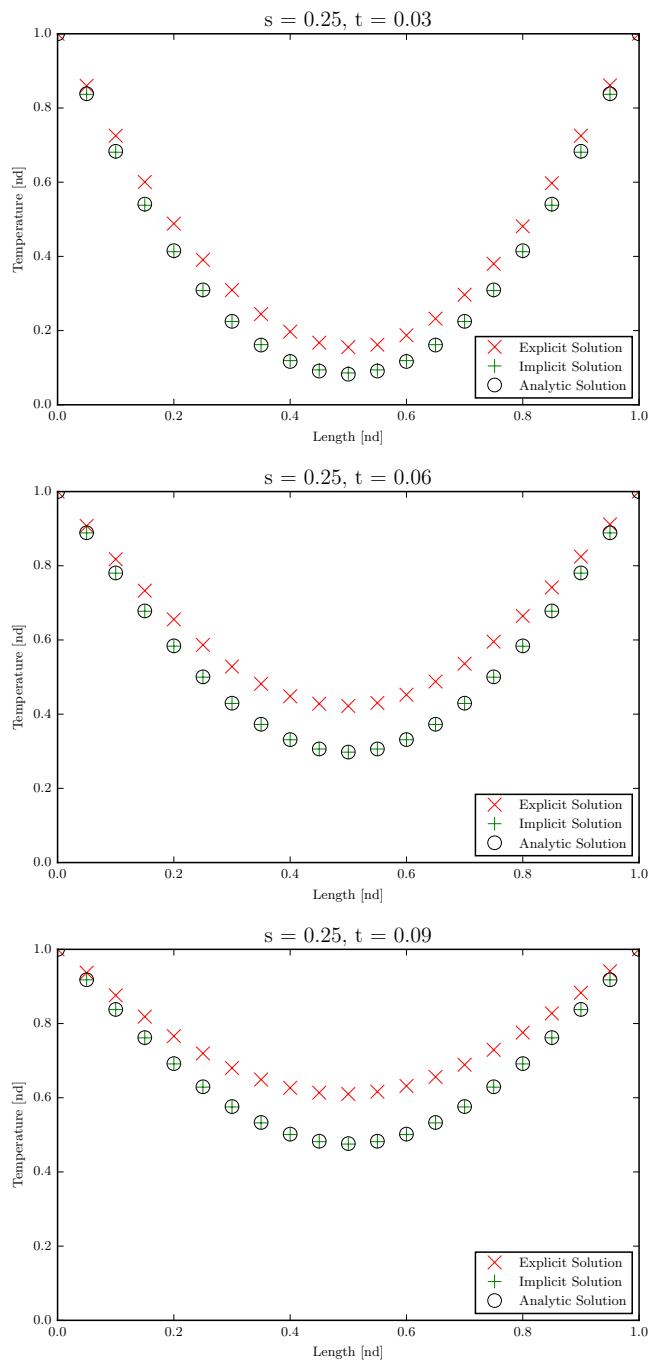


Fig. 2. Results for $s = 0.25$

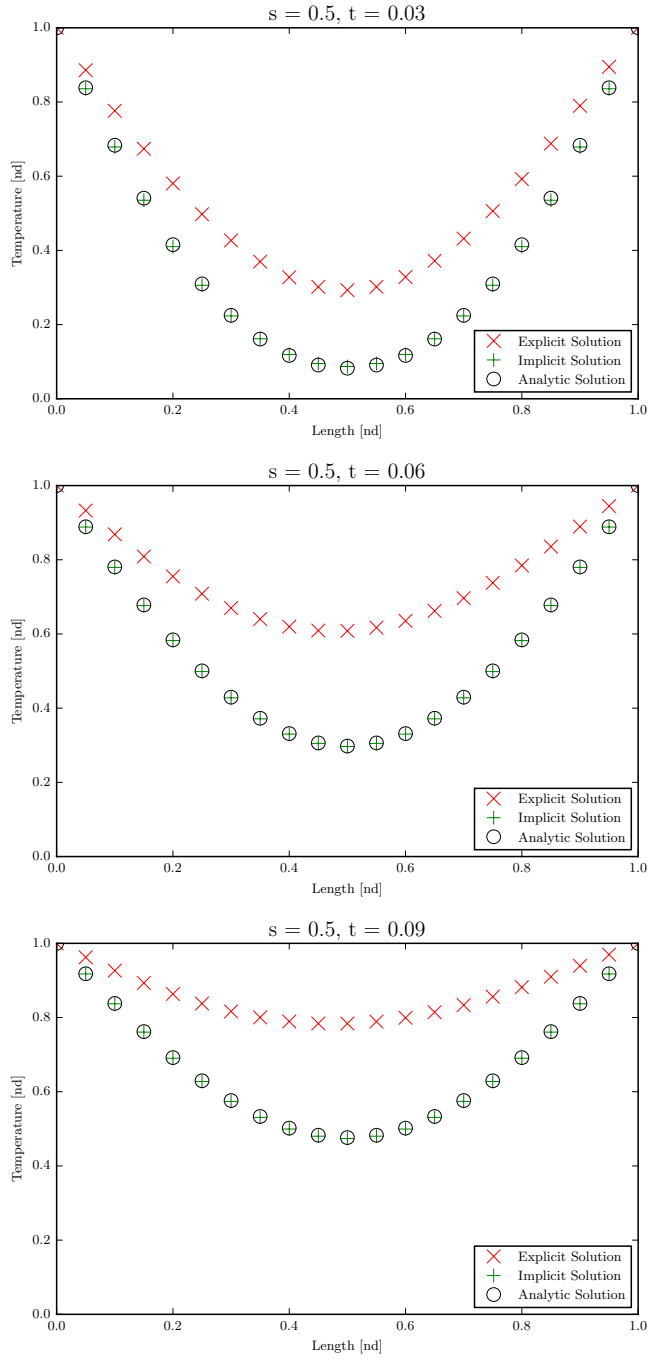


Fig. 3. Results for $s = 0.5$

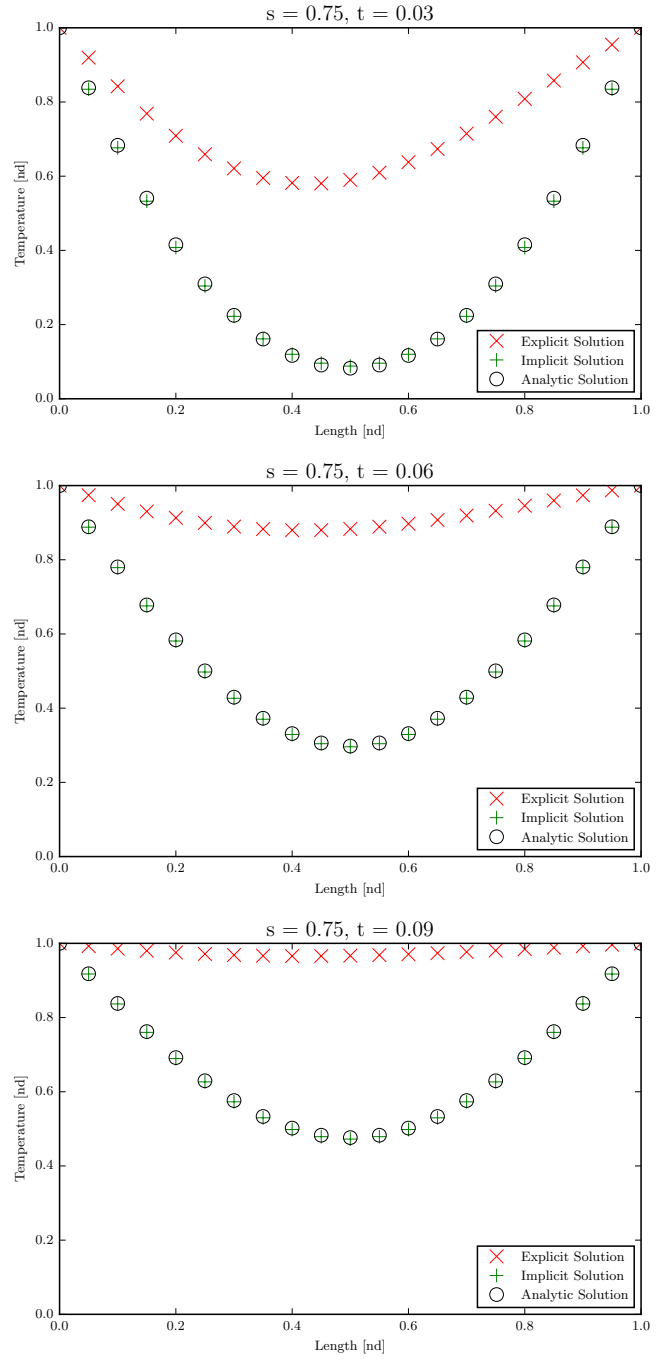


Fig. 4. Results for $s = 0.75$

Appendix A: Python Code

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import os
4
5 # Configure figures for production
6 WIDTH = 495.0 # the number latex spits out
7 FACTOR = 1.0 # the fraction of the width you'd like the figure to occupy
8 fig_width_pt = WIDTH * FACTOR
9
10 inches_per_pt = 1.0 / 72.27
11 golden_ratio = (np.sqrt(5) - 1.0) / 2.0 # because it looks good
12
13 fig_width_in = fig_width_pt * inches_per_pt # figure width in inches
14 fig_height_in = fig_width_in * golden_ratio # figure height in inches
15 fig_dims = [fig_width_in, fig_height_in] # fig dims as a list
16
17
18 def Solver(s, t_end, show_plot=False):
19     # Problem Parameters
20     L = 1. # Domain length [n.d.]
21     T0 = 0. # Initial temperature [n.d.]
22     T1 = 1. # Boundary temperature [n.d.]
23     N = 21
24
25     # Set-up Mesh
26     x = np.linspace(0, L, N)
27     dx = x[1] - x[0]
28
29     # Calculate time-step
30     dt = s * dx ** 2.0
31
32     # Initial Condition with boundary conditions
33     T_initial = [T0] * N
34     T_initial[0] = T1
35     T_initial[N - 1] = T1
36
37     # Explicit Numerical Solution
38     T_explicit = Explicit(np.array(T_initial).copy(), t_end, dt, s)
39
40     # Implicit Numerical Solution
41     T_implicit = Implicit(np.array(T_initial).copy(), t_end, dt, s)
42
43     # Analytical Solution
44     T_analytic = np.array(T_initial).copy()
45     for i in range(0, N):
46         T_analytic[i] = Analytic(x[i], t_end)
47
48     # Find the RMS
49     RMS = RootMeanSquare(T_implicit, T_analytic)
50     ExplicitRMS = RootMeanSquare(T_explicit, T_analytic)
51
52     # Format our plots
53     plt.figure(figsize=fig_dims)
54     plt.axis([0, L, T0, T1])
55     plt.xlabel('Length [nd]')
56     plt.ylabel('Temperature [nd]')
57     plt.title('s = ' + str(s)[:5] + ', t = ' + str(t_end)[:4])
58
59     # ...and finally plot
60     plt.plot(x, T_explicit, 'xr', markersize=9, label='Explicit Solution')
61     plt.plot(x, T_implicit, '+g', markersize=9, label='Implicit Solution')
62     plt.plot(x, T_analytic, 'ob', markersize=9, mfc='none', label='Analytic Solution')
63     plt.legend(loc='lower right')
64
65     # Save plots
66     save_name = 'proj_1_s_' + str(s)[:5] + '_t_' + str(t_end) + '.pdf'
```

```

67     try:
68         os.mkdir('figures')
69     except Exception:
70         pass
71
72     plt.savefig('figures/' + save_name, bbox_inches='tight')
73     if show_plot:
74         plt.show()
75     plt.clf()
76
77     return RMS, ExplicitRMS
78
79
80 def Explicit(Told, t_end, dt, s):
81     """
82     This function computes the Forward-Time, Centered-Space (FTCS) explicit
83     scheme for the 1D unsteady heat diffusion problem.
84     """
85     N = len(Told)
86     time = 0.
87     Tnew = Told
88
89     while time <= t_end:
90         for i in range(1, N - 1):
91             Tnew[i] = s * Told[i + 1] + (1 - 2.0 * s) * Told[i] + s * Told[i - 1]
92
93         Told = Tnew
94         time += dt
95
96     return Told
97
98
99 def Implicit(Told, t_end, dt, s):
100     """
101     This function computes the Forward-Time, Centered-Space (FTCS) implicit
102     scheme for the 1D unsteady heat diffusion problem.
103     """
104     N = len(Told)
105     time = 0.
106
107     # Build our 'A' matrix
108     a = [-s] * N
109     a[0], a[-1] = 0, 0
110     b = [1 + 2 * s] * N
111     b[0], b[-1] = 1, 1 # hold boundary
112     c = a
113
114     while time <= t_end:
115         Tnew = TDMASolver(a, b, c, Told)
116
117         Told = Tnew
118         time += dt
119
120     return Told
121
122
123 def RootMeanSquare(a, b):
124     """
125     This function will return the RMS between two lists (but does no checking
126     to confirm that the lists are the same length).
127     """
128     N = len(a)
129
130     RMS = 0.
131     for i in range(0, N):
132         RMS += (a[i] - b[i]) ** 2.
133

```

```

134     RMS = RMS ** (1. / 2.)
135     RMS /= N
136
137     return RMS
138
139
140 def TDMAsolver(a, b, c, d):
141     """
142     Tridiagonal Matrix Algorithm (a.k.a Thomas algorithm).
143     """
144     N = len(a)
145     Tnew = d
146
147     # Initialize arrays
148     gamma = np.zeros(N)
149     xi = np.zeros(N)
150
151     # Step 1
152     gamma[0] = c[0] / b[0]
153     xi[0] = d[0] / b[0]
154
155     for i in range(1, N):
156         gamma[i] = c[i] / (b[i] - a[i] * gamma[i - 1])
157         xi[i] = (d[i] - a[i] * xi[i - 1]) / (b[i] - a[i] * gamma[i - 1])
158
159     # Step 2
160     Tnew[N - 1] = xi[N - 1]
161
162     for i in range(N - 2, -1, -1):
163         Tnew[i] = xi[i] - gamma[i] * Tnew[i + 1]
164
165     return Tnew
166
167
168 def Analytic(x, t):
169     """
170     The analytic answer is 1 - Sum(terms). Though there are an infinite
171     number of terms, only the first few matter when we compute the answer.
172     """
173     result = 1
174     large_number = 1E6
175
176     for k in range(1, int(large_number) + 1):
177         term = ((4. / ((2. * k - 1.) * np.pi)) *
178                np.sin((2. * k - 1.) * np.pi * x) *
179                np.exp(-(2. * k - 1.) ** 2. * np.pi ** 2. * t))
180
181         # If subtracting the term from the result doesn't change the result
182         # then we've hit the computational limit, else we continue.
183         # print '{0} {1}, {2:.15f}'.format(k, term, result)
184         if result - term == result:
185             return result
186         else:
187             result -= term
188
189
190 def main():
191     """
192     Main function to call solver over assigned values and create some plots to
193     look at the trends in RMS compared to s and t.
194     """
195     # Loop over requested values for s and t
196     s = [1. / 6., .25, .5, .75]
197     t = [0.03, 0.06, 0.09]
198
199     RMS = []
200     with open('results.dat', 'w+') as f:

```

```

201     for i, s_ in enumerate(s):
202         sRMS = [0] * len(t)
203         for j, t_ in enumerate(t):
204             sRMS[j], ExplicitRMS = Solver(s_, t_, False)
205             f.write('{0:.3f} {1:.2f} {2:.2e} {3:.2e} \n'.format(s_, t_, sRMS[j], ExplicitRMS))
206             # print i, j, sRMS[j]
207         RMS.append(sRMS)
208
209     # Convert to np array to make this easier...
210     RMS = np.array(RMS)
211
212     # Check for trends in RMS vs t
213     plt.figure(figsize=fig_dims)
214     plt.plot(t, RMS[0], '.r', label='s = 1/6')
215     plt.plot(t, RMS[1], '.g', label='s = .25')
216     plt.plot(t, RMS[2], '.b', label='s = .50')
217     plt.plot(t, RMS[3], '.k', label='s = .75')
218     plt.xlabel('t')
219     plt.ylabel('RMS')
220     plt.title('RMS vs t')
221     plt.legend(loc='best')
222
223     save_name = 'proj_1_rms_vs_t.pdf'
224     plt.savefig('figures/' + save_name, bbox_inches='tight')
225     plt.clf()
226
227     # Check for trends in RMS vs s
228     plt.figure(figsize=fig_dims)
229     plt.plot(s, RMS[:, 0], '.r', label='t = 0.03')
230     plt.plot(s, RMS[:, 1], '.g', label='t = 0.06')
231     plt.plot(s, RMS[:, 2], '.b', label='t = 0.09')
232     plt.xlabel('s')
233     plt.ylabel('RMS')
234     plt.title('RMS vs s')
235     plt.legend(loc='best')
236
237     save_name = 'proj_1_rms_vs_s.pdf'
238     plt.savefig('figures/' + save_name, bbox_inches='tight')
239     plt.clf()
240
241 if __name__ == "__main__":
242     main()

```