In this project we are going to use numerical integration combined with linear algebra to solve integral equations. The scripts and functions you will be asked to produce are all quite short; but they are tricky to write and debug. You will also learn more about passing functions and their arguments to other functions in Matlab.

**Fredholm integral equations**

A Fredholm integral equation has the form

\[ g(s) = \int_0^1 K(s, t) f(t) \, dt - \mu f(s). \tag{1} \]

The function \( K(k, t) \) is called the kernel of the equation. Given a function \( g(s) \) we want to determine the function \( f(t) \). If \( \mu \) is zero, (1) is a Fredholm equation of the first kind; otherwise it is an equation of the second kind. Equations of the first kind have many important applications: computer aided tomography and image deblurring among others. But they are generally hard to solve. Here we will consider equations of the second kind, which are usually more tractable.

**A strategy for computing an approximation**

Only in very simple cases is it possible to find an analytic solution of (1). Instead we will compute an approximation to the solution by replacing the integral with an integration formula. The process can be illustrated by the by the trapezoidal.

We begin by dividing the interval \([0, 1]\) into \( n \) equally spaced points, \( s_1 = t_1, s_2 = t_2, \ldots, s_n = t_n \). Set

\[ g_i = g(s_i), \]

If we knew the values of \( f \) exactly we would have

\[ g_i = \int_0^1 K(s_i, t) f(t) \, dt - \mu f(s_i). \tag{2} \]

Since we do not, we try to determine approximations \( f_i \) to \( f(t_i) \), that satisfy

\[ g_i = C T^1_0 K(s_i, t) f(t) - \mu f_i, \]

where \( CT \) is the composite trapezoidal rule on the points \( t_1, \ldots, t_n \).

Let \( h = 1/(n-1) \) be the distance between consecutive \( t_i \). Then replacing the integral in (2) by the composite trapezoidal rule, we get the equation

\[ g_i = \frac{h}{2} (K(s_i, t_1) f_1 + 2K(s_i, t_2) f_2 + \cdots + 2K(s_i t_i, t_{n-1}) f_{n-1} + K(s_i, t_n) f_n) - \mu f_i. \]

\( ^1 \)The notation used here is slightly nonstandard.
This equation must hold for \( i = 1, \ldots, n \). This gives us \( n \) linear equations, which may be solved for the \( f_i \).

The matrix form for the system system is easily exhibited. Let \( k_{ij} = K(s_i, t_j) \). Then we have for \( n = 5 \)

\[
\begin{pmatrix}
 g_1 \\
 g_2 \\
 g_3 \\
 g_4 \\
 g_5
\end{pmatrix} = \frac{h}{2} \begin{pmatrix}
 k_{11} - \frac{2\mu}{h} & 2k_{12} & 2k_{13} & 2k_{14} & k_{15} \\
 k_{21} & 2k_{22} - \frac{2\mu}{h} & 2k_{23} & 2k_{24} & k_{25} \\
 k_{31} & 2k_{32} & 2k_{33} - \frac{2\mu}{h} & 2k_{34} & k_{35} \\
 k_{41} & 2k_{42} & 2k_{43} & 2k_{44} - \frac{2\mu}{h} & k_{45} \\
 k_{51} & 2k_{52} & 2k_{53} & 2k_{54} & k_{55} - \frac{2\mu}{h}
\end{pmatrix} \begin{pmatrix}
 f_1 \\
 f_2 \\
 f_3 \\
 f_4 \\
 f_5
\end{pmatrix}
\]

Thus our algorithm is to evaluate the \( g_i \), form the matrix in (3), and use the Matlab operator \( \setminus \) to solve the system for the \( f_i \).

### Generating test cases

One of the difficulties in debugging any scientific program is generating test cases with known solutions. This is particularly so of our problem, since it will in general be impossible to integrate (2) analytically. Of course, we could use numerical integration to generate the \( g_i \), but that would create errors of its own.

We are going to use numerical integration, but in such a way that it is exact. Specifically, our kernel will be

\[
K_p(s, t) = \left(1 - (s - t)^2\right)^p
\]

and we will let \( f_t \) be a low order polynomial; e.g., \( 1 + t^2 \). Since for fixed \( s \), \( K_p(s, t) \) is a polynomial of degree \( 2p \) in \( t \). Hence if the degree of \( f \) is \( q \), then a Gaussian quadrature whose order \( \ell \) satisfies \( 2\ell - 1 \geq 2p + q \) will integrate (2) exactly.

The first step is to code the following function.

```matlab
function intgrl = gauss24(a, b, fun, varargin)
% GAUSS24  Gauss-Legendre integration with 24 points.
% INTGRL = GAUSS24(A, B, FUN, VARARGIN) returns a 24 point
% Gauss-Legendre approximation to the integral over the
% interval [A,B] of the function FUN. FUN is a string
% containing the name of the function to be evaluated. It’s
% calling sequence (within GAUSS24) is
% VAL = FEVAL(FUN, T, VARARGIN(:));
% FUN may return a vector of values, in which case GAUSS24
% will return a vector of integrals.
```
Note that the specifications are in Matlab format, which requires that variables and functions be written in capital letters for emphasis. They should be reduced to lower case in your code. The ordinates and abscissas for the quadrature will be provided.

The argument varargin is one solution to a problem that arises frequently in scientific computation. The utility function gauss24 must evaluate a function. Its name is provided by the argument fun, and gauss24 will generate the values t at which it is to be evaluated. However, fun requires more information — for example, the order p of the kernel. One solution would be to include the additional information in the calling sequence of gauss24; e.g.,

```matlab
intgrl = gauss24(0, 1, 'myfunction', p);
```

But then gauss24 would not work with another function that did not require p or required more than one additional argument. Matlab’s solution is to collect all the final arguments after ‘myfunction’ into a cell array and pass it to gauss24. When gauss24 invokes feval to evaluate fun, Matlab unpacks varargin and places its objects at the end of the calling sequence of fun. For further details, use the help command.

To evaluate the $g_i$ in (2) you must write two further functions:

```matlab
function y = kerf(t, f, s, p)
    % KERF(T, F, S, P) is called by gauss24 to evaluate
    % the integrand of the integral equation (1) (in
    % the writeup for Project 3) for the scalar T and
    % the vector S. F is the name of the function in
    % the integrand and P is the order of the kernel (4).
    % The calling sequence for F is
    %
    %    FEVAL(F, T);

    %
    %
    %
end
```

and

```matlab
function g = ggen(s, f, mu, p)
    % GGEN(S, F, MU, P) generates the vector G defined by (2) in the
    % writeup for Project 3. S is the vector of S(i), F is the name
    % of the function f in the kernel with calling sequence
    %
    %    FEVAL(F, T);
    %
    %    MU is as in equation (2) and P is the order of the kernel.

    %
    %
end
```

The body of each function can be written in a single line of code.
Setting up the equations.
We have now generated a vector $g$ that was evaluated at the points in the vector $s$ for the function $f$. We must now generate the matrix in (3), but for general $n$. Write the following function.

```matlab
function K = Ktrap(n, p, mu)
    % K = KTRAP(N, P, MU) sets up the matrix of the system
    % (3) in the Project 3 writup. P is the order of the kernel,
    % MU is as in (3).
```

In our experiments, we want to compare the trapezoidal rule with Simpson’s rule. Code the following function.

```matlab
function K = Ksimp(n, p, mu)
    % K = KSIMP(N, P, N) is the analogue of KTRAP but with
    % the compound Simpson’s rule replacing the trapezoidal rule.
    % An error return is made if N is even.
```

You will find your life easier if you first write out the matrix as in (2).

**The bottom line**
In writing your code you should try to avoid for loops. You will be much helped by the Matlab . operator that implements elementwise operations. For example, the loop

```matlab
for i=1:n
    f(i) = 1 + 2*x(i)^2;
end
```

can be replaced by

```matlab
f = 1 + 2*x.^2;
```

The fewer loops in your code, the faster it will run and the better your grade will be.

Run your routines with the script in Figure 1. It may help you in debugging to know that the plot should be two straight lines with negative slopes.

Turn listings of your functions, the plot, and the answer to the following question. What are numbers `mysterynumtrap` and `mysterynumsimp` and how do they relate to the errors in the composite trapezoidal and Simpson’s rules?
k = 20;
mu = -1;

% Compute the trapezoidal approximations and plot the error.
nrm = [];
for n = 21:10:101
    s = linspace(0, 1, n)';
    f = fun(s);
    g = ggen(s, 'fun', mu, k);
    K = Ktrap(n, k, mu);
    ft = K\g;
    nrm = [nrm, norm(ft - f)/norm(f)];
end
loglog(21:10:101,nrm, '-')
hold
nrm
[m,k] = size(nrm);
mysterynumtrap = log10(nrm(k)/nrm(1))/log10(101/21)

% Compute the Simpson approximations and plot the error.
nrm = [];
for n = 21:10:101
    s = linspace(0, 1, n)';
    f = fun(s);
    g = ggen(s, 'fun', mu, k);
    K = Ksimp(n, k, mu);
    ft = K\g;
    nrm = [nrm, norm(ft - f)/norm(f)];
end
loglog(21:10:101,nrm,'.-')
nrm
[m,k] = size(nrm);
mysterynumsimp = log10(nrm(k)/nrm(1))/log10(101/21)
hold

Figure 1: Script for Project 3