Illuminating Numerical Analysis using Mathematica

Jennifer Voitle and Edward Lumsdaine

Abstract

This paper discusses the experiences of the authors teaching courses in Numerical Analysis to engineering students at two universities, both pre- and post-introduction of ***Mathematica***. Use of ***Mathematica*** in such courses really enhances and accelerates student learning and comprehension, providing a foundation for success in subsequent courses. With ***Mathematica***, more realistic and complex applications can be assigned, since ***Mathematica*** has many of the required intrinsic functions to solve real-life problems. Tips are given to increase student acceptance of ***Mathematica***, as some students will resist it for various reasons.

This paper also discusses the use of ***Mathematica*** as a programming language, including transition from and communication with other languages. Much of the course can be taught using the intrinsic functions available i***n Mathematica***, but for certain applications extension of the built-in capabilities is required. For example, in earthquake and vibration engineering, systems of differential equations must be solved whose forcing functions are given as tabular data. This paper shows how to read in these data from a file and manipulate them so that **NDSolve** can work with them.

One of the most powerful ways to use ***Mathematica*** as a teaching tool is via ***Mathematica*** Notebooks. The course can be taught electronically - as a "live" textbook. Students can obtain the needed information from these Notebooks, which include text, algorithms, animations and problem sets. The paper discusses such Notebooks to teach Numerical Analysis, including guidelines for effective design. Where desired, assignments can be given in the "traditional" way of writing computer programs by following algorithms, but us***ing Mathematica*** as the programming language. Other assignments can be given which involve the use or modification of existing ***Mathematica*** functions. Freed from the tedious program writing, testing, debugging and compiling of programs, this method encourages students to explore "what-if" questions by altering parameters and methods.

Mathematica as a Programming Language

Mathematica is a rule-based language. Transition from other languages is easy, as ***Mathematica*** provides capabilities for all of the standard constructs, eg. looping, conditional testing, etc. The development time and size of a ***Mathematica*** program will generally be substantially less than in traditional languages such as FORTRAN, BASIC, C and so forth.

Summing Integers

As an example, consider the problem of summing integers from 1 to some ending value. That is, we seek

$$\sum_{i=1}^{100} i$$

The FORTRAN code to accomplish this objective is

```
SUM = 0
    WRITE(*,*) ' ENTER ENDING VALUE TO SUM TO : '
    READ(*,*) N
    DO 10 I = 1, N
     SUM = SUM + I
10     CONTINUE
    WRITE(*,*) SUM
    END
```

Similar code would be written in BASIC:

```
sum = 0
INPUT "enter ending value to sum to:", n
FOR i = 1 TO n
  sum = sum + i
NEXT
PRINT sum
```

or in C:

```
/* sumint.c -- program to sum from 1 to n */
#include <stdio.h>

int main (void)
{
    long int i, n, sum;
    sum = 0;
    printf ("Enter ending value to sum to:\n");
    scanf ("%ld", &n);
    for (i = 1; i <= n; i++)
      sum += i;
    printf ("The sum is %ld.", sum);
    return 0;
}
```

Mathematica code

In *Mathematica*, this program can be written in a single line,since there is a Sum function built in. The upper index n is passed as a function argument. Suppose that we choose n=100:

```
100
∑ i
i=1
```

```
5050
```

One unfortunate fact regarding other programming languages is that they only have finite precision. ***Mathematica*** has infinite precision, which means that one could ask for the sum of i from one to one billion and the correct answer would be obtained, an impossible feat in other languages. However, the way that ***Mathematica*** does the sum makes the result, while correct, prohibitively slow. To speed things up, change the way the sum is evaluated. ***Mathematica*** has a **Range** function, which outputs a list ranging over the limits provided by the user. For example, **Range[10]** produces the following:

```
Range[10]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Mathematica has another function, **Plus**, which takes a set of numbers as its input and returns their sum.

For example:

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

```
55
```

To avoid having to type the arguments to **Plus**, combine it with **Range** using the command **Apply** as follows:

```
Plus @@ Range[10]
```

```
55
```

Apply works by taking the output form of **Range**, which was a list of numbers, and replacing this by **Plus**. So the output is the sum of the given set of numbers. To appreciate this, compute the sum of integers from 1 to 10000 using both **Sum** and **Apply** with **Range**. Wrap the function **Timing** around these for comparison. I am running this on a Macintosh SE/30 running an unenhanced copy of ***Mathematica***. Your times will vary.

```
$Version
```

```
4.0 for Microsoft Windows (July 26, 1999)
```

```
Timing[Sum[i, {i, 10000}]]
```

```
{0.05 Second, 50005000}
```

```
Timing[Plus @@ Range[10000]]
```

```
{0. Second, 50005000}
```

Fixed Point Iteration

Solve x = g(x) given an initial guess p0, a tolerance tol, and a maximum number of iterations itmax.

BASIC:

```
'*************** user's function goes here ***************
DEF fng(x) = (x+1)^(1/3)
'*****************************************************
tol = 0.0001
itmax = 100
p0 = 1.5     ' initial guess
i = 1
status$= "iterating"
PRINT"    Solution of x = g(x) by Fixed Point Iteration"
PRINT"iteration     x       g(x) "
WHILE (i<=itmax AND status$ <>"Converged")
      pnew=fng(p0)
      PRINT i,p0,pnew
      IF ABS(pnew-p0)<=tol THEN
          status$ = "Converged"
      END IF
       p0 = pnew
       i = i +1
WEND

IF status$ = "Converged" THEN
   PRINT"Method converged in ",i -1
   PRINT" iterations to root ",pnew
ELSE
   PRINT"Method did not converge in ",itmax," iterations."
END IF
```

C:

```
/ * fixedpoint.c -- program to perform fixed point iteration * / # include <
 stdio.h > # include < math.h > # define TOL 0.0001
# define ITMAX 100
# define CONVERGED 1
# define ITERATING 0
double g (double x); / * function prototype * /

  int main (void)
{
 double p0, pnew;
 int i = 1;
 int status = ITERATING;
 printf ("Fixed Point Iteration\n");
 printf ("Please enter an initial guess:\n");
 scanf ("%lf", & p0);
 printf ("\nIteration\tx\t\tg(x)\n");
 while (i ≤ ITMAX && status ≠ CONVERGED) {pnew = g (p0);
    printf ("%d\t\t%6.8lf\t\t %6.8lf\n", i, p0, pnew);
    if ((fabs (pnew - p0)) ≤ TOL) status = CONVERGED;
    p0 = pnew;
    i ++;} // end of while block if (status == CONVERGED)
   {printf ("\nMethod converged in %d iterations.", i);
    printf ("\nRoot is %6.8lf", pnew);} else printf
   ("\nMethod did not converge after %d iterations.\n", ITMAX);
 return 0;
}

double g (double x)
{
   return pow ((x + 1), (1. / 3));   // users function definition
}
```

Mathematica as a C (or BASIC) interpreter:

```
Clear[g]
g[x_] = (x + 1)^(1/3);
tol = 0.0001;
itmax = 100;
p0 = 1.5;
i = 1;
status = "iterating"
Print["    Solution of x = ", g[x], " by Fixed Point Iteration"];
Print["iteration    x       g(x) "];
While[i ≤ itmax && status ≠ "Converged",
 pnew = g[p0]; Print[i, "\t\t\t", p0, "\t", pnew];
 If[Abs[pnew - p0] ≤ tol, status = "Converged"]; p0 = pnew; i += 1]
If[status === "Converged",
 Print["Method converged in ", i -= 1, " iterations to root ", pnew],
 Print["Method did not converge in ", itmax, " iterations."]]
```

the *Mathematica* programming language

Of course, ***Mathematica*** provides the functions **NestList** and **FixedPoint**:

```
Clear[g]
g[x_] = (x + 1)^(1/3);
itmax = 5;
p0 = 1.5;
NestList[g, p0, itmax]
```

```
{1.5, 1.35721, 1.33086, 1.32588, 1.32494, 1.32476}
```

Rootfinding

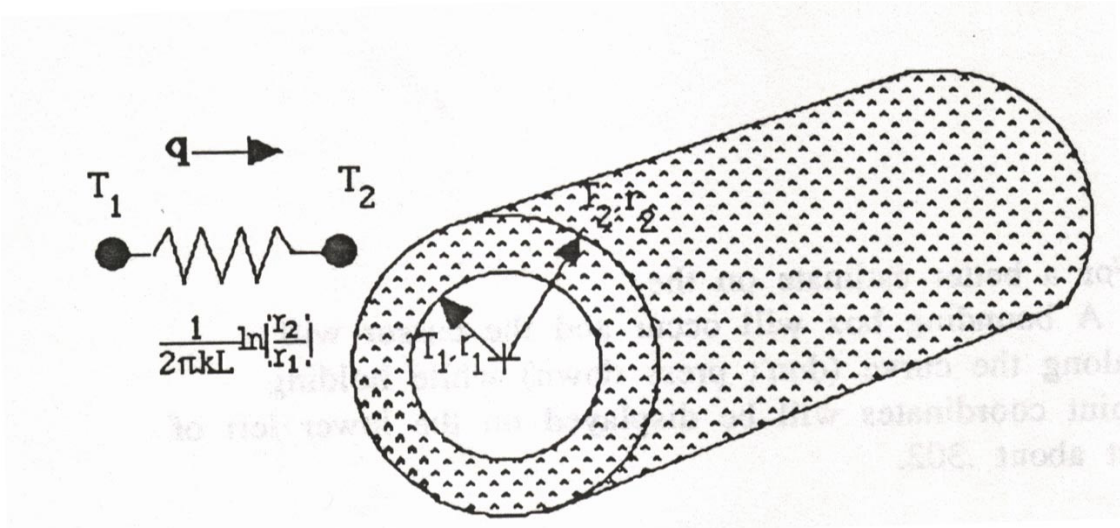
Example: Heat Transfer through Concentric Cylinders

The analysis of heat transfer through a series of concentric cylinders requires the solution of the following equation. Determine the thickness Δr = r2 - r1 of the inner cylinder given that

$$\ln\left(\frac{r_2}{r_1}\right) = 2.2185 \ln\left(\frac{0.2640}{r_2 \ln\left(\frac{r_2}{r_1}\right)}\right)$$

by all root-finding methods. Note that r2 and r1 are in meters. Take r1 = 0.1575 m. Summarize results. Did Steffensen's method result in any improvement?

Schematic



Analysis

The given equation must be solved for r2. r1 is given as 0.1575 m. It would be reasonable to expect that r2 > r1 but r2< 1, say. The solution will be carried out by the Bisection Method, Fixed Point iteration, Steffensen's Method, Newton's Method and the Secant Method. The solution will be carried out to a tolerance of 0.001 and all results compared.

Solution

Function Definition

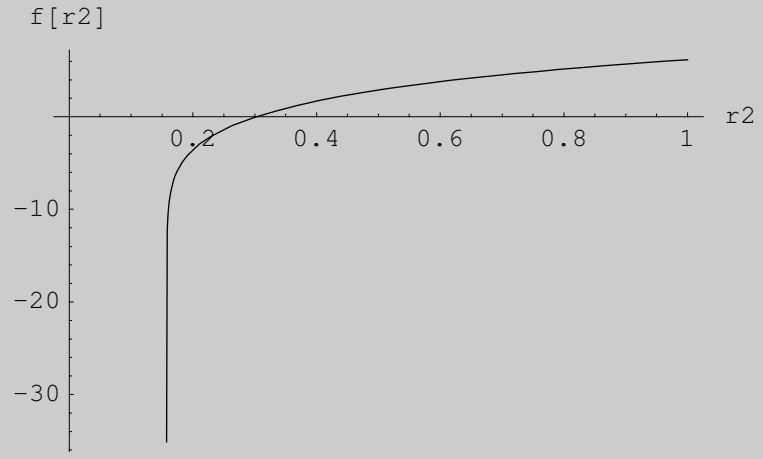
Define r1 and and f[r2]:

```
r1 = 0.1575;
f[r2_] = Log[r2/r1] - 2.2185 Log[.264/(r2 Log[r2/r1])];
```

Graphical Estimation of Root

Use the Plot command to plot the function for a graphical estimation of the root. We could plot the left hand side against the right hand side and look for an intersection of the two curves, or set the right hand side to zero and look for roots of the function. It is the latter approach that we use here:

```
Plot[f[r2], {r2, r1, 1}, AxesLabel → {"r2", "f[r2]"}]
```



The root is at approximately r2 = 0.3. For a better estimate on the Macintosh, click anywhere on the graph. A bounding box will occur and the cursor will change to a compass. Move the cursor along the curve (don't press down) while holding down the Command (Apple) key. The point coordinates will be displayed on the lower left of the screen. By this method, the root is at about .302.

Solution by Solve

Mathematica has the **Solve** function, which works really well only for polynomials and simpler non-polynomial equations. It will not work on our particular function, which is why numerical methods are necessary.

```
N[Solve[f[r2] == 0, r2]]
```

```
— Solve::tdep : The equations appear to involve
     the variables to be solved for in an essentially non-algebraic way.

— Solve::tdep : The equations appear to involve
     the variables to be solved for in an essentially non-algebraic way.
```

$$\text{Solve}\left[\text{Log}[6.34921\ r2] - 2.2185\ \text{Log}\left[\frac{0.264}{r2\ \text{Log}[6.34921\ r2]}\right] == 0.,\ r2\right]$$

■ Bisection Method

This example uses our package, bisection.m, described in the appendix. For now, just use the function definition:

```
Bisection[f_, var_, {LeftEndPoint_, RightEndPoint_}, tol_, itmax_] :=
 Module[{iteration, func}, a = N[LeftEndPoint]; b = N[RightEndPoint];
  func = Function[var, f];  Print["i\ta\tb\tp\tf[a]\tf[b]\tf[p]"];
  For[iteration = 1, iteration <= itmax, iteration++,
   (While[func[a] func[b] > 0, (Message[Bisection::nosignchange];
      {a, b} = Input["Enter new values {a,b}:"])];
    MidPoint = N[(b + a) / 2];
    Print[iteration, "\t", N[a, 4], "\t", N[b, 4], "\t", N[MidPoint, 6],
     "\t", N[func[a], 4], "\t", N[func[b], 4], "\t", N[func[MidPoint], 4]];
    If[func[a] * func[MidPoint] > 0, a = MidPoint, b = MidPoint];
    If[Abs[func[MidPoint]] <= tol, Break[]];)];
  If[iteration <= itmax, Message[Bisection::converged, iteration - 1],
   Message[Bisection::unconverged, itmax]];
  Print["\nThe root is found approximately as \n", N[MidPoint]];
  Print[" where the function value is ", func[MidPoint], "."];]
```

```
Bisection[f[r2], r2, {.2, .4}, .001, 10]
```

```
i    a    b    p    f[a]    f[b]    f[p]

1    0.2    0.4    0.3    -3.55336    1.69772    -0.0470806

2    0.3    0.4    0.35    -0.0470806    1.69772    0.924903

3    0.3    0.35    0.325    -0.0470806    0.924903    0.470302

4    0.3    0.325    0.3125    -0.0470806    0.470302    0.220581

5    0.3    0.3125    0.30625    -0.0470806    0.220581    0.0891619

6    0.3    0.30625    0.303125    -0.0470806    0.0891619    0.0216669

7    0.3    0.303125    0.301563
     -0.0470806    0.0216669    -0.0125472

8    0.301563    0.303125    0.302344
     -0.0125472    0.0216669    0.00459939

9    0.301563    0.302344    0.301953
     -0.0125472    0.00459939    -0.00396397

10    0.301953    0.302344    0.302148
     -0.00396397    0.00459939    0.000320186
```

— *Bisection::converged :*
The Bisection Method converged to the root of the function in 9 iterations.

```
The root is found approximately as
0.302148
```

```
 where the function value is 0.000320186.
```

■ Solution by Fixed Point Iteration

The given equation has to be reformulated in the form r2 = g[r2].

This may most easily be done by adding r2 to both sides of the equation:

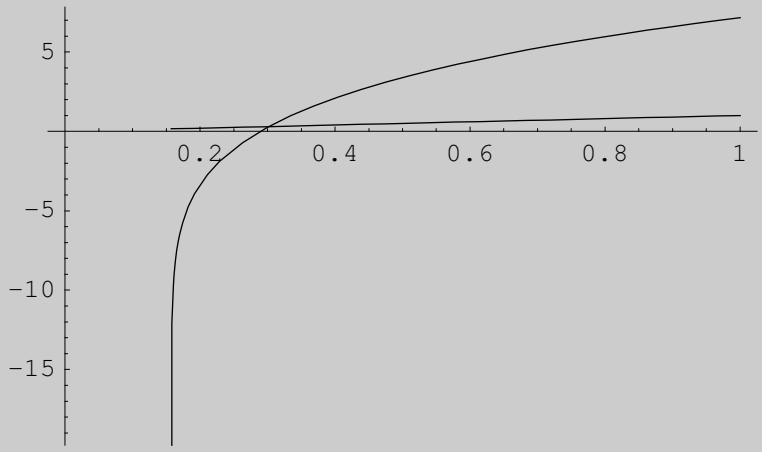
□ Trial 1

```
Clear[g]
g[r2_] = f[r2] + r2
```

$$r2 + \text{Log}[6.34921\ r2] - 2.2185\ \text{Log}\left[\frac{0.264}{r2\ \text{Log}[6.34921\ r2]}\right]$$

Plot to see if we expect convergence:

```
Plot[{r2, g[r2]}, {r2, r1, 1}]
```



It is clear from the plot that this choice of g will cause the method to diverge. A few iterations will verify this:

```
NestList[g, .4, 5]
```

```
{0.4, 2.09772, 11.3956, 27.2563, 46.335, 67.3389}
```

This is a case of monotone divergence. Different formulations of g can be attempted. In this case, convergence will be attained if we solve for the innermost r2 in the nested logs. We find:

$$r_2 = r_1 \, Exp\left(\frac{\frac{0.2640}{\left(\frac{r_2}{r_1}\right)^{\frac{1}{2.2185}}}}{r_2} \right)$$

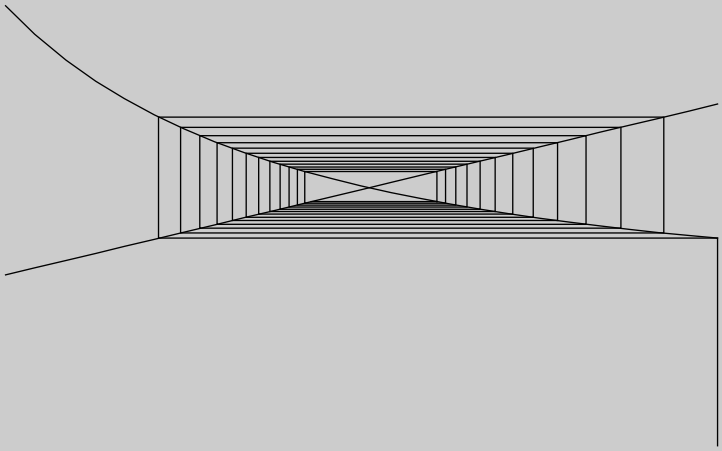
```
Clear[g]

g[r2_] = r1 e^(.264 / (r2/r1)^(1/2.2185) / r2)
```

```
0.1575 e^(0.114756 / r2^1.45076)
```

Twenty five iterations of the fixed point method and a plot of the path are generated by the code below:

```
ShowFixedPath[g_Symbol, initguess_, numiterations_, {a_, b_}] :=
 Module[{graph1, lines, x},
  iterates = NestList[g, initguess, numiterations]; lines = Partition[
    Flatten[Table[{iterates[[i]], iterates[[i]]}, {i, Length[iterates]}]],
    2, 1]; lines = lines /. lines[[1]] → {lines[[1, 1]], 0};
  graph1 = Plot[{x, g[x]}, {x, a, b}, DisplayFunction → Identity];
  FixedPointPath = Graphics[Line[lines]]; Show[FixedPointPath,
   graph1, DisplayFunction → $DisplayFunction]; iterates]
```

```
ShowFixedPath[g, .4, 25, {.2, .4}]
```



```
{0.4, 0.242991, 0.384908, 0.249113, 0.372905, 0.254543, 0.363136,
 0.259386, 0.355046, 0.263721, 0.348254, 0.267613, 0.342492,
 0.271111, 0.337561, 0.27426, 0.333314, 0.277095, 0.329634, 0.279648,
 0.326432, 0.281947, 0.323634, 0.284018, 0.321183, 0.285881}
```

This method appears to be exhibiting spiral convergence. It will take a large amount of iterations to converge since the slope of g[r2] approaches unity (the upper asymptote for convergence) near the root.

```
Abs[g'[r2]] /. r2 → .305
```

```
0.915321
```

In fact, fixed point iteration will take 208 iterations to converge to the root of 0.302134. Either NestList or FixedPoint can be used to illustrate this:

```
{FixedPoint[g, .4, 207], FixedPoint[g, .4, 208]}
```

```
{0.302133, 0.302134}
```

■ Steffensen's Algorithm

To increase the speed of convergence of the fixed point method,
compute p1 = g[p0], p2 = g[p1]. So far, this is identical to the fixed point method. But now compute

$$p = p_0 - \frac{(p_1 - p_0)^2}{p_2 - 2p_1 + p_0}$$

where p is assumed to be a better estimate of the root than p0, p1 or p2. Convergence is attained when |p - p0| <= tol. If not, p0 is set equal to p and the process is continued.
This algorithm can be easily programmed in ***Mathematic***a as:

```
Steffensen[x_List] := x[[1]] - (x[[2]] - x[[1]])^2 / (x[[3]] - 2 x[[2]] + x[[1]])
```

Start by generating the required estimates p1, p2 by NestList. These will be used by Steffensen to get the next estimate p:

□ Iteration 1

```
x = NestList[g, .4, 2]
```

```
{0.4, 0.242991, 0.384908}
```

```
p = Steffensen[x]
```

```
0.317532
```

□ Iteration 2

Now use this as p0 to generate p1 and p2:

```
x = NestList[g, p, 2]
```

```
{0.317532, 0.288745, 0.315811}
```

The improved estimate p is

```
p = Steffensen[x]
```

```
0.302695
```

□ Iteration 3

```
x = NestList[g, p, 2]
```

```
{0.302695, 0.301605, 0.302635}
```

```
p = Steffensen[x]
```

```
0.302135
```

□ Iteration 4

```
x = NestList[g, p, 2]
```

```
{0.302135, 0.302133, 0.302135}
```

```
p = Steffensen[x]
```

```
0.302134
```

This is the solution. ***It took four iterations compared to the two hundred and eight required by fixed point, dramatically demonstrating the acceleration of convergence.***

A simple Steffensen program can be written as follows:

```
x = NestList[g, .4, 2];
SteffensenTable = {};
Do[p = Steffensen[x]; x = NestList[g, p, 2];
 SteffensenTable = Append[SteffensenTable, p], {k, 5}]
SteffensenTable
```

```
{0.317532, 0.302695, 0.302135, 0.302134, 0.302134}
```

■ Solution by Newton-Raphson Method

The Newton-Raphson method uses the scheme

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

to approximate the solution of the equation f(x) = 0.

Mathematica has the built-in function **FindRoot** to approximate the solution of nonlinear equations and systems of equations by the Newton-Raphson and the Secant methods. The syntax is:

```
FindRoot[lhs == rhs, {var,guess}] performs rootfinding by Newton-Raph-
son iteration to approximate the solution of the
equation lhs = rhs using the initial estimate guess  for the unknown
var.

FindRoot[lhs == rhs, {var,{guess1,guess2}}] performs rootfinding by
Secant iteration using two initial estimates guess1  and  guess2  for
the unknown var.

FindRoot[{lhs1==rhs1,lhs2==rhs2,...,lhsn==rhsn}, {var1,guess1},
```

{var2,guess2}, ..., {varn,guessn}] returns the numerical approximation of the system of equations with given initial estimates guess1, guess2, ... for the unknowns.

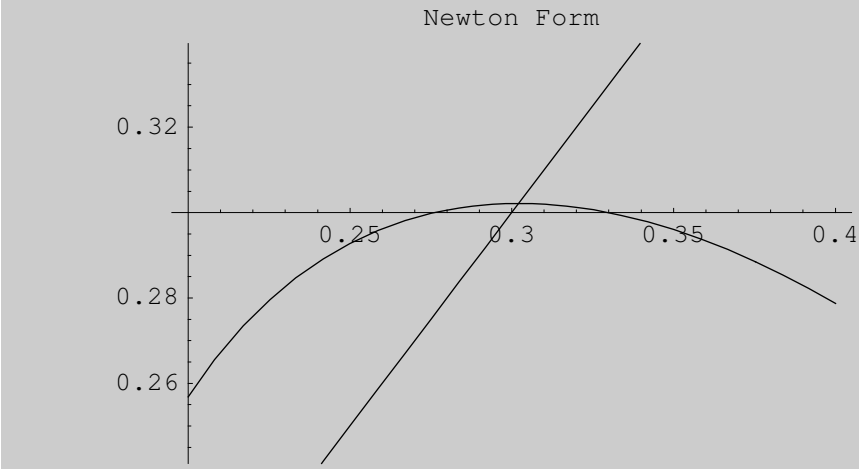
```
FindRoot[f[r2] == 0, {r2, .4}]
```

```
{r2 → 0.302134}
```

To see the intermediate calculations, **NestList** can be used. Newton's method is a special case of fixed point iteration, with g defined as follows:

```
Clear[g]
g[r2_] = r2 - f[r2]/f'[r2]
Plot[{r2, g[r2]}, {r2, 0.2`, 0.4`}, PlotLabel → "Newton Form"]
```

$$r2 - \frac{\text{Log}[6.34921\ r2] - 2.2185\ \text{Log}\left[\frac{0.264}{r2\ \text{Log}[6.34921\ r2]}\right]}{\frac{1.}{r2} - 8.40341\ r2\left(-\frac{0.264}{r2^2\ \text{Log}[6.34921\ r2]^2} - \frac{0.264}{r2^2\ \text{Log}[6.34921\ r2]}\right)\text{Log}[6.34921\ r2]}$$



It is clear that this scheme will converge. **NestList** will show the calculations:

```
NestList[g, .4, 10]
```

```
{0.4, 0.278707, 0.300405, 0.302125, 0.302134,
 0.302134, 0.302134, 0.302134, 0.302134, 0.302134, 0.302134}
```

Convergence attained in five iterations with 0.4 as the initial guess.

■ Solution by the Secant Method

The Secant method is a variation of Newton's method which replaces the analytical derivative f'[x] by a numerical approximation

$$f'(x) \approx \frac{f(x+h)-f(x)}{h}$$

Thus, the secant scheme is

$$x_{i+2} = x_{i+1} - \frac{f(x_{i+1})(x_{i+1}-x_i)}{f(x_{i+1})-f(x_i)}, \quad i = 0,1,2,..., done$$

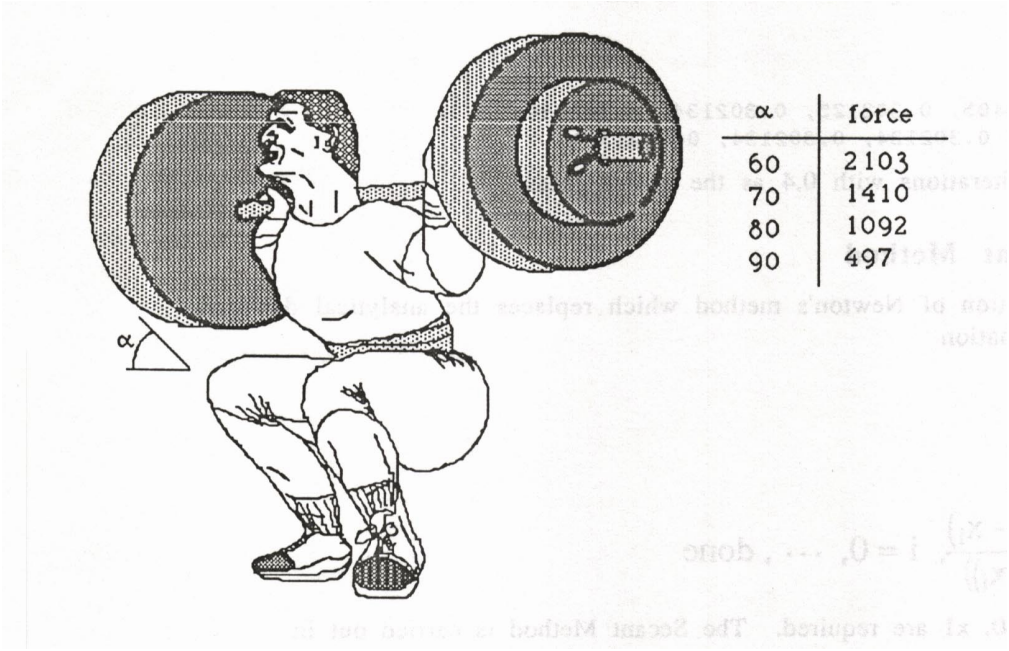
Note that two initial guesses x0, x1 are required. The Secant Method is carried out in ***Mathematica*** by calling **FindRoot** with two initial guesses:

```
FindRoot[f[r2] == 0, {r2, {0.2, 0.4}}]
```

```
{r2 → 0.302134}
```

Function Interpolation and Approximation

Example: Force on a Weightlifter's Back



Polynomial Interpolation

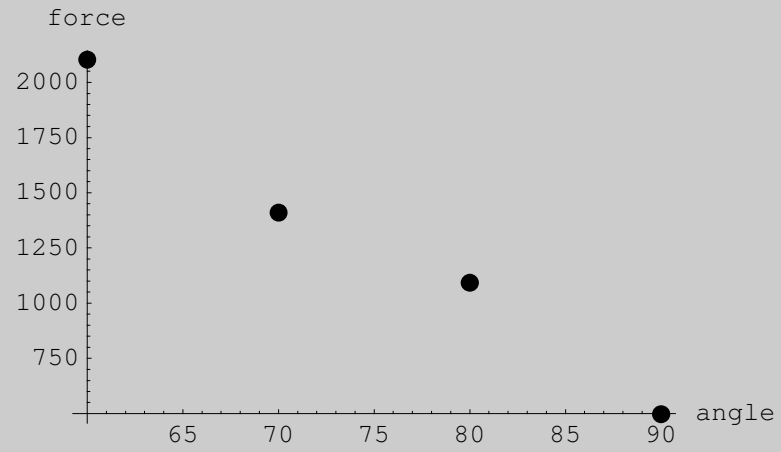
The problem may be solved by fitting an interpolating polynomial through the (angle,force) data. This approximates the function force(angle) which is then solved for the angle giving a force of 1250 pounds. Alternatively, inverse interpolation may be performed. In this case, an inverse function angle(force) is generated which may then be evaluated at 1250. ***Mathematica*** has the function **InterpolatingPolynomial** to perform these tasks. The output is identical to the expanded Lagrange or Newton interpolating polynomial. A short program, **NewtonDividedDifference**, is presented to show the intermediate steps of the calculations.

■ Data Definition and Plot

```
force[60] = 2103; force[70] = 1410;
force[80] = 1092; force[90] = 497;
WeightLifterData = Table[{i, force[i]}, {i, 60, 90, 10}]
```

```
{{60, 2103}, {70, 1410}, {80, 1092}, {90, 497}}
```

```
DataPlot = ListPlot[WeightLifterData,
  PlotStyle → PointSize[0.03`], AxesLabel → {"angle", "force"}]
```

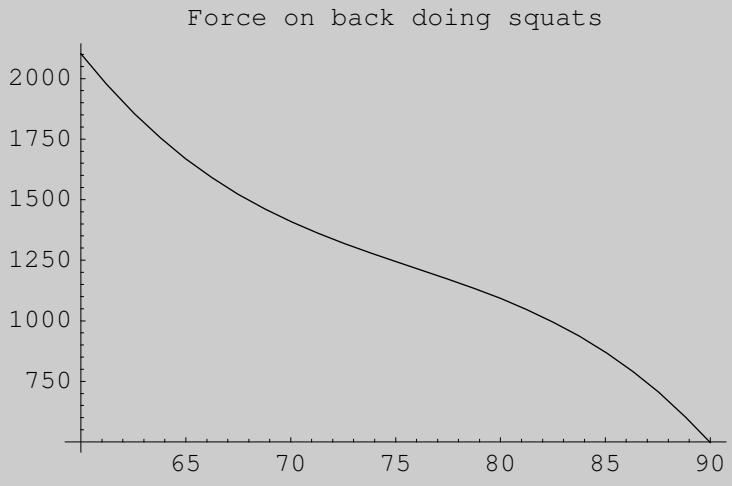


■ Lagrange Form of Interpolating Polynomial

```
P3[angle_] = Expand[InterpolatingPolynomial[WeightLifterData, angle]]
```

$$50\,648 - \frac{22\,795\,\text{angle}}{12} + \frac{4939\,\text{angle}^2}{200} - \frac{163\,\text{angle}^3}{1500}$$

```
Plot[P3[angle], {angle, 60, 90}, PlotLabel → "Force on back doing squats"]
```



The above graph shows that an angle of approximately 74 degrees will result in a force of 1250 pounds on the back. We will verify this with FindRoot:

```
FindRoot[P3[x] == 1250, {x, 74}]
```

```
{{0.302134, 0.302134, 0.302134} → 74.8241}
```

```
Print["The force exerted on the back while squatting at an angle of ",
 %[[1, 2]], " degrees is ", P3[x] /. %, " pounds."]
```

```
The force exerted on the back while squatting at an angle of
 74.8241 degrees is {50 076.3, 50 076.3, 50 076.3} pounds.
```

Interpolation by Newton's Divided Differences

■ Description of Procedure NewtonDD

Procedure NewtonDD takes a list of n+1 {x,y} pairs as its input and returns a interpolating polynomial of degree n in the variable *var*.

```
Clear[NewtonDD, x]

NewtonDD[data_List, var_Symbol] :=
 Module[{i, n, poly, xx, f},
  xx = var;
  Do[xx[i - 1] = data[[i, 1]];
   f[i - 1] = data[[i, 2]], {i, Length[data]}];
  f[i_, 0] := f[i];
  (f[n_, i_] := (f[n, i - 1] - f[n - 1, i - 1]) / (xx[n] - xx[n - i]) /; i > 0);
  Sum[f[i, i] Product[var - xx[j - 1], {j, i}], {i, 0, Length[data] - 1}] ]
```

```
NewtonDD[WeightLifterData, x]
```

$$2103 - \frac{693}{10}(-60+x) + \frac{15}{8}(-70+x)(-60+x) - \frac{163(-80+x)(-70+x)(-60+x)}{1500}$$

In this form, the coefficients a0, a1, a2, ... can easily be identified for checking hand calculations. Expand this to verify that this form is identical to the Lagrange polynomial:

```
Expand[%] === P3[x]
```

```
True
```

■ Solution by Inverse Interpolation

Here we construct a function angle(force) by interchanging the (angle,force) data. This function is simply evaluated at force = 1250 for the desired angle.

```
angle[force_] = N[Expand[
   InterpolatingPolynomial[Table[{force[i], i}, {i, 60, 90, 10}], force]]]
```

$$73.9901 + 0.0656302\,\text{force} - 0.0000774083\,\text{force}^2 + 2.04647 \times 10^{-8}\,\text{force}^3$$

```
angle[1250]
```

```
75.0472
```

This compares reasonably well to the previous answer.

Cubic Spline Interpolation

■ Solution by Natural Cubic Splines

Mathematica has a built-in spline function, but it does not give the spline equations for evaluation. Here, we use our own spline package to construct S[0,x] on the interval [60,70]; S[1,x] on [70,80], and S[2,x] on [80,90]. To locate the angle at which the force is equal to 1250 pounds, note that f[80] = 1092 and f[70] = 1410. Since
1092 < 1250 < 1410, the desired angle is between 70 and 80 degrees. Thus we will perform rootfinding on S[1,x] = 1250.
The natural cubic spline uses the boundary conditions S"[0,x0]=0 and S"[n-1,xn] = 0.

Here, we import our spline interpolation package, which we have stored in the standard package library in the folder NumericalMath. (Of course, *Mathematica* now has its own spline routines.)

```
Needs["NumericalMath`SplineInterpolation`"]
```

```
Information["SplineInterpolation", LongForm → False]
```

```
The package SplineInterpolation contains code for  construction of
  natural, clamped, periodic and B cubic splines. A plotting utility,
  PiecewiseCubicPlot, and integration procedure, IntegrateSpline,
  are also provided.  For information on usage of these procedures,
  type ?NaturalCubicSpline, ?ClampedCubicSpline, ?PeriodicSpline,
  ?BSplineInterpolation, ?PiecewiseCubicPlot or ?IntegrateSpline.
```

```
Information["NaturalCubicSpline", LongForm → False]
```

```
NaturalCubicSpline[{{x0,y0},{x1,y1},...{xn,yn}}]   constructs and prints
  a natural cubic spline interpolating the given   data points. The
  natural spline boundary conditions   S''[x0] = S''[xn]=0 are used.
```

```
NaturalCubicSpline[WeightLifterData]
```

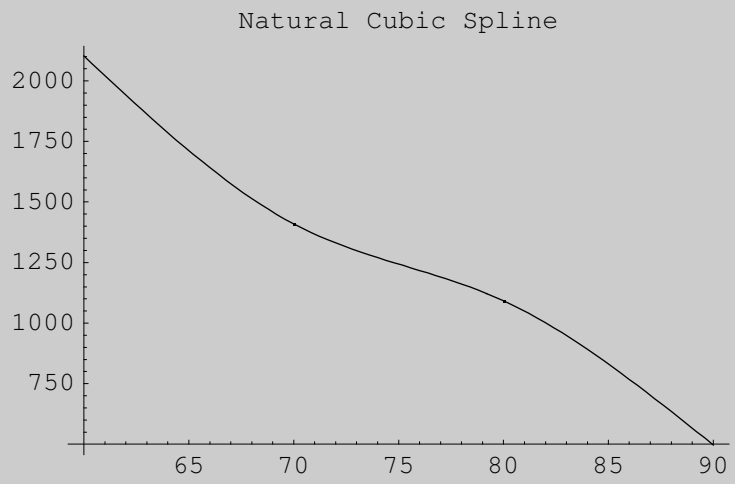
The spline is constructed as follows:

$$S[0,x] = -18\,617. + 1198.29\,x - 21.324\,x^2 + 0.118467\,x^3 \quad \text{for } 60. < x < 70.$$

$$S[1,x] = 96\,562.4 - 3737.97\,x + 49.194\,x^2 - 0.217333\,x^3 \quad \text{for } 70. < x < 80.$$

$$S[2,x] = -65\,332. + 2333.07\,x - 26.694\,x^2 + 0.0988667\,x^3 \quad \text{for } 80. < x < 90.$$

```
PiecewiseCubicPlot[PlotLabel → "Natural Cubic Spline"];
```




```
S[1, x_] = 96562.4 - 3737.97 x + 49.194 x^2 - 0.217333 x^3;
N[Solve[S[1, x] == 1250, x]]
```

```
{{x → 74.7558}, {x → 75.7987 - 11.0028 i}, {x → 75.7987 + 11.0028 i}}
```

■ Solution by Clamped Cubic Splines

The package SplineInterpolation.m also has a utility for construction of clamped cubic splines, which match the function derivatives at the endpoints x0, xn. Then S'[0,x0] = f'[x0], S'[n-1,xn] = f'[xn]. For this example involving tabular data, we shall approximate the derivatives numerically using three-point forward and backward difference formulas. These formulas are:

$$f'(x_i) = \frac{-3f(x_i)+4f(x_{i+1})-f(x_{i+2})}{2h} \quad forward\ \ difference\ \ O(h^2)$$

$$f'(x_i) = \frac{3f(x_i)-4f(x_{i-1})+f(x_{i-2})}{2h} \quad backward\ difference\ \ O(h^2)$$

The ***Mathematica*** code to perform these derivatives is:

```
Clear[FD3Pt, BD3Pt]
FD3Pt[f_List, h_] := (-3 f[[1]] + 4 f[[2]] - f[[3]])/(2 h)
BD3Pt[f_List, h_] := (3 Last[f] - 4 f[[Length[f] - 1]] + f[[Length[f] - 2]])/(2 h)
```

```
- General::spell1 :
   Possible spelling error: new symbol name "BD3Pt" is similar to existing symbol "FD3Pt".
```

These equations can be used to approximate our endpoint derivatives as follows:

```
Derivs = N[{FD3Pt[{2103, 1410, 1092}, 10], BD3Pt[{1410, 1092, 497}, 10]}]
```

```
{-88.05, -73.35}
```

The procedure **ClampedCubicSpline** requires the given data as input and these derivatives. The usage is:

```
Information["ClampedCubicSpline", LongForm → False]
```

```
ClampedCubicSpline[{{x0,y0},{x1,y1},...{xn,yn},{y'0,y'n}}]
  constructs and prints a clamped cubic spline interpolating
  the given   data points. The clamped spline boundary
  conditions   S'[x0] = f'[x0],S'[xn] = f'[xn] are used.
```

Modify the weight lifter data to include the derivatives using **Append**:

```
Append[WeightLifterData, Derivs]
```

```
{{60, 2103}, {70, 1410}, {80, 1092}, {90, 497}, {-88.05, -73.35}}
```

ClampedCubicSpline[%]

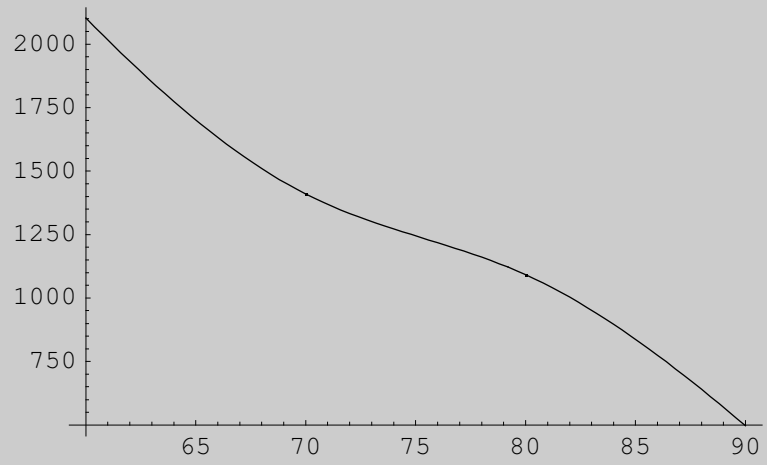
The spline is constructed as follows:

$$S[0,x] = -2294.4 + 469.35\,x - 10.513\,x^2 + 0.0652\,x^3 \quad \text{for } 60. < x < 70.$$

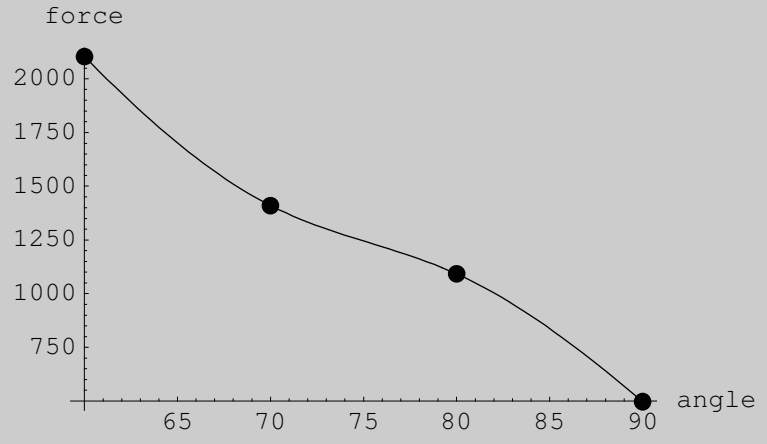
$$S[1,x] = 87160. - 3364.41\,x + 44.255\,x^2 - 0.1956\,x^3 \quad \text{for } 70. < x < 80.$$

$$S[2,x] = -46369.6 + 1642.95\,x - 18.337\,x^2 + 0.0652\,x^3 \quad \text{for } 80. < x < 90.$$

PiecewiseCubicPlot[];



Show[DataPlot, %]



```
N[Solve[87160. - 3364.41 x + 44.255 x^2 - 0.1956 x^3 == 1250, x]]
```

```
{{x → 74.8099}, {x → 75.7213 - 11.7188 i}, {x → 75.7213 + 11.7188 i}}
```

Function Approximation by Regression

Regression can be performed in ***Mathematica*** by either the Fit function or by loading the Statistics`LinearRegression. This package computes the best fit polynomial as does Fit, but also prints a complete report of the results including an ANOVA table.

■ The Regression Fits and Roots

```
Do[func = Fit[WeightLifterData, Table[angle^i, {i, 0, k}], angle];
 result = Solve[func == 1250, angle][[1]];
 Print[k, "\t", func, "\t", result], {k, 1, 3}]
```

```
1    5127.5 - 51.36 angle    {angle → 75.4965}
```

```
2    6475. - 88.11 angle + 0.245 angle^2    {angle → 74.9003}
```

```
3    50648. - 1899.58 angle + 24.695 angle^2 - 0.108667 angle^3
     {angle → 74.8241}
```

```
N[P3[angle]]
```

```
50648. - 1899.58 angle + 24.695 angle^2 - 0.108667 angle^3
```

Note that the cubic regression polynomial is identical to the cubic generated by InterpolatingPolynomial.
(WHY?)
What general conclusions can you draw from this?

Numerical Solution of Differential Equations

Application

(Dr. Naser Mostaghel's problem)

In earthquake engineering, the following system of ODE's requires solution:

$$\frac{dz}{dt} + 2\alpha\xi\beta z(t) + \alpha^2 \Omega^2 u(t) = -u_g(t)$$

$$\frac{du}{dt} = z(t)$$

$$z(t_0) = y'(t_0),\ y(t_0) = y_0,\ t_0 \le t \le t_{\max}$$

where ug(t) represents the displacement of the ground with time and u(t) is the resulting displacement of the structure under consideration. Although ***Mathematica***'s intrinsic functions **DSolve** and **NDSolve** should be able to solve this system, the difficulty in solving this particular problem is that ug(t) is usually discrete data points which must be read in from a data file. Currently, **DSolve** and **NDSolve** require continuous inputs, so some manipulation of the data must be performed before these functions see it.

In the following example, we shall first assume that ug(t) is well-represented by a sine function and will apply **DSolve** directly to the problem. We then read in the actual data from the file, manipulate it into a continuous form and compare the results from **DSolve**ing this to the preceding results.

■ Parameter Definitions

```
Clear[u, z, omega, alpha, beta, y0, y0prime, u, z, t, xi]
xi = 0.05; omega = 2. π; alpha = 1.5; beta = 1.0;
t0 = 0.48843; yprime0 = 18.1134; y0 = 1.0;
tend = 6; a = 240; delta = 2.5 omega;
ug[t_] = a Sin[delta t];
```

Solution with ug[t] = a Sin[δ t]:

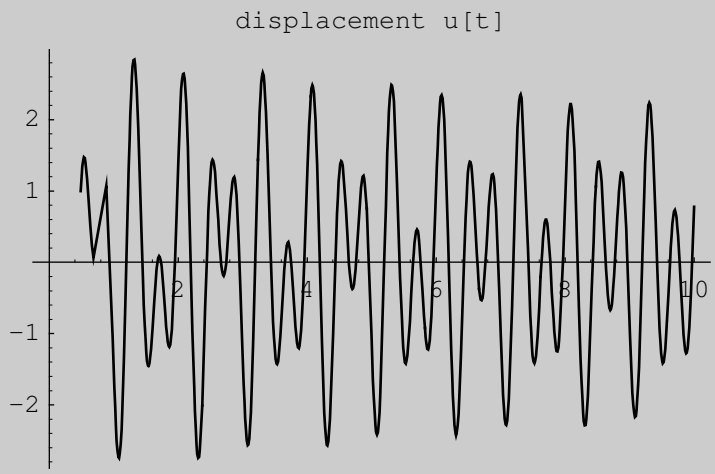
Mathematica cannot solve this! This is unfortunately what often happens with highly complex real problems...

```
Clear[u, z]
solution =
 Flatten[DSolve[{z'[t] + 2 alpha beta xi z[t] + alpha^2 omega^2 u[t] == -ug[t],
    u'[t] == z[t], z[t0] == yprime0, u[t0] == y0}, {z[t], u[t]}, t]]
```

```
— FactorSquareFree::lrgexp : Exponent is out of bounds for function FactorSquareFree.

— PolynomialGCD::lrgexp : Exponent is out of bounds for function PolynomialGCD.

— FactorSquareFree::lrgexp : Exponent is out of bounds for function FactorSquareFree.

— FactorSquareFree::lrgexp : Exponent is out of bounds for function FactorSquareFree.

— General::stop :
   Further output of FactorSquareFree::lrgexp will be suppressed during this calculation.

— PolynomialGCD::lrgexp : Exponent is out of bounds for function PolynomialGCD.

— PolynomialGCD::lrgexp : Exponent is out of bounds for function PolynomialGCD.

— General::stop :
   Further output of PolynomialGCD::lrgexp will be suppressed during this calculation.
```

Had ***Mathematica*** had been able to solve this, we would have plotted using the following command:

```
plot1 = Plot[Evaluate[u[t] /. solution],
  {t, t0, tend}, PlotLabel → "displacement u[t]"]
```



Solution with ug[t] read in from a file

The data file, collected from data acquisition software, is shown below. The data are stored in the file "quakedata".

```
0.48843     236.435
3.20602     22.9008
5.92361     -223.666
8.6412      -144.339
10.         0.0304909
```

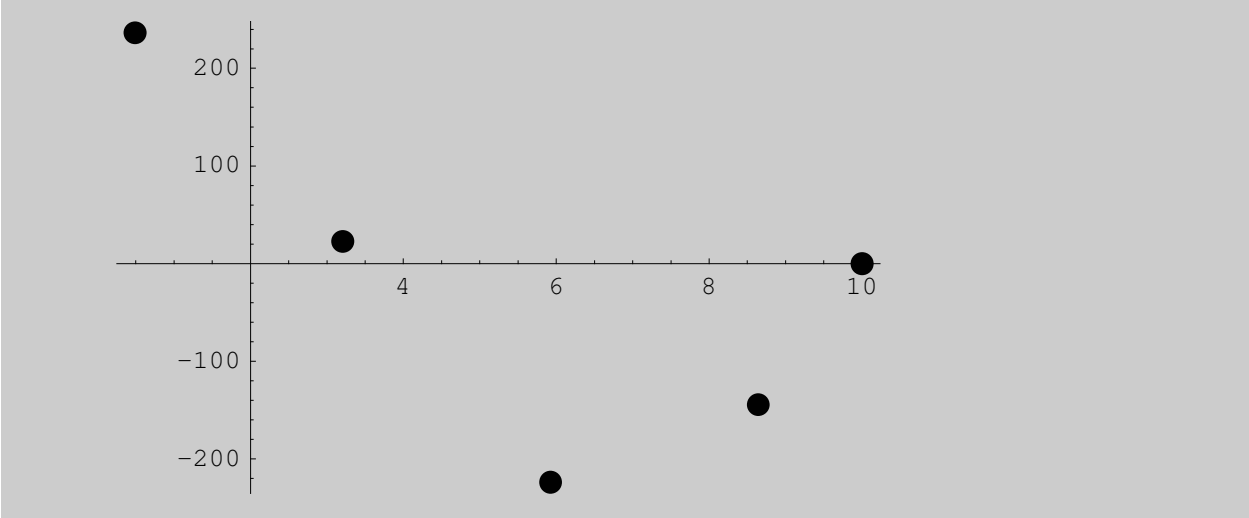
All that needs to be done is to read in the data in the proper form for ***Mathematica*** manipulation, that is, in the form
{{pair1},{pair2}, ... ,{pairn}}. This can be accomplished with the **OpenRead** command (here, it is assumed that the data are contained in the ***Mathematica*** folder. If not, the required information would be included.)

```
OpenRead["quakedata.txt"]
GroundData = ReadList["quakedata.txt", {Number, Number}]
```

```
InputStream[quakedata.txt, 17]
```

```
{{0.48843, 236.435}, {3.20602, 22.9008},
 {5.92361, -223.666}, {8.6412, -144.339}, {10., 0.0304909}}
```

```
inputdataplot = ListPlot[GroundData, PlotStyle → PointSize[0.03`]]
```



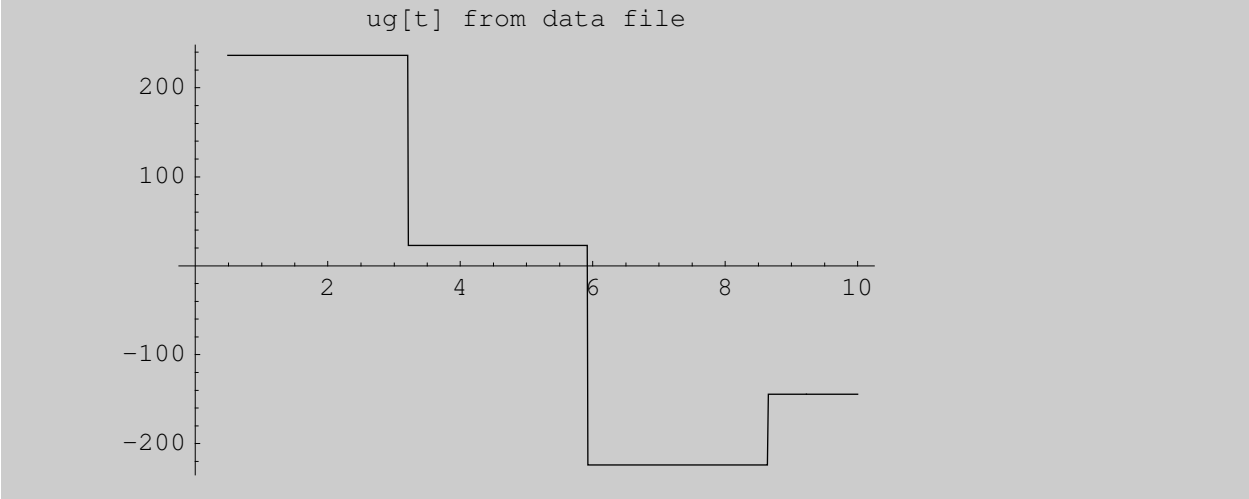
A curve-fit of the data (regression, interpolating polynomial, spline, etc) could be attempted. Please refer back to the previous section if this is desired.

(For example, we could type **Fit[GroundData,{Sin[N[delta] t]},t]** to obtain the result 240.1597 Sin[15.7079 t]).

Here, we will construct a piecewise-continuous function using **Which**. First, put the data into the proper form using the **Table** command to automate things. This creates a list. Then Which is mapped onto the list, which is named ug[t].

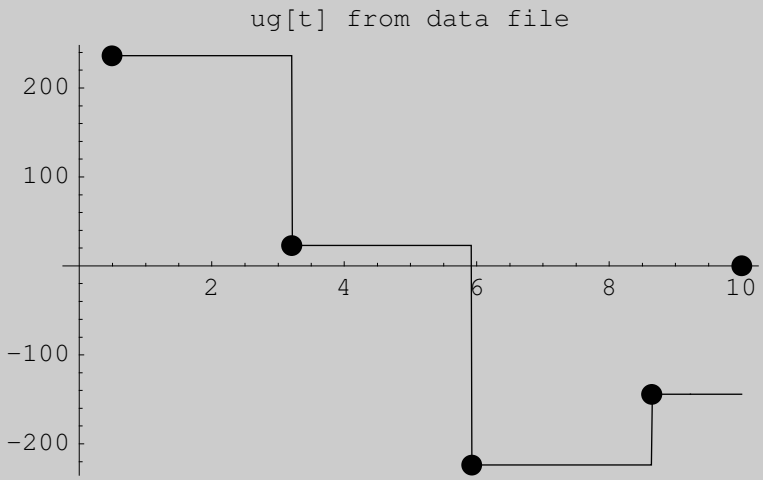
```
Clear[ug]
ug[t_] := Which @@ Flatten[Append[
    Table[{t ≥ GroundData⟦i, 1⟧ && t < GroundData⟦i + 1, 1⟧, GroundData⟦i, 2⟧},
     {i, 1, Length[GroundData] - 1}], {t ≥ GroundData⟦Length[GroundData], 1⟧,
     GroundData⟦Length[GroundData], 2⟧}]]
```

```
Plot[ug[t], {t, GroundData⟦1, 1⟧, GroundData⟦Length[GroundData], 1⟧},
 PlotLabel → "ug[t] from data file"]
```



Comparison of actual data and Which approximation

```
Show[%, inputdataplot]
```



Note that if this approximation is unsatisfactory, (perhaps one desires the points to be at the midpoint of the interval rather than at the left endpoints), this is easily accomplished by modification of the code used to construct ug[t].

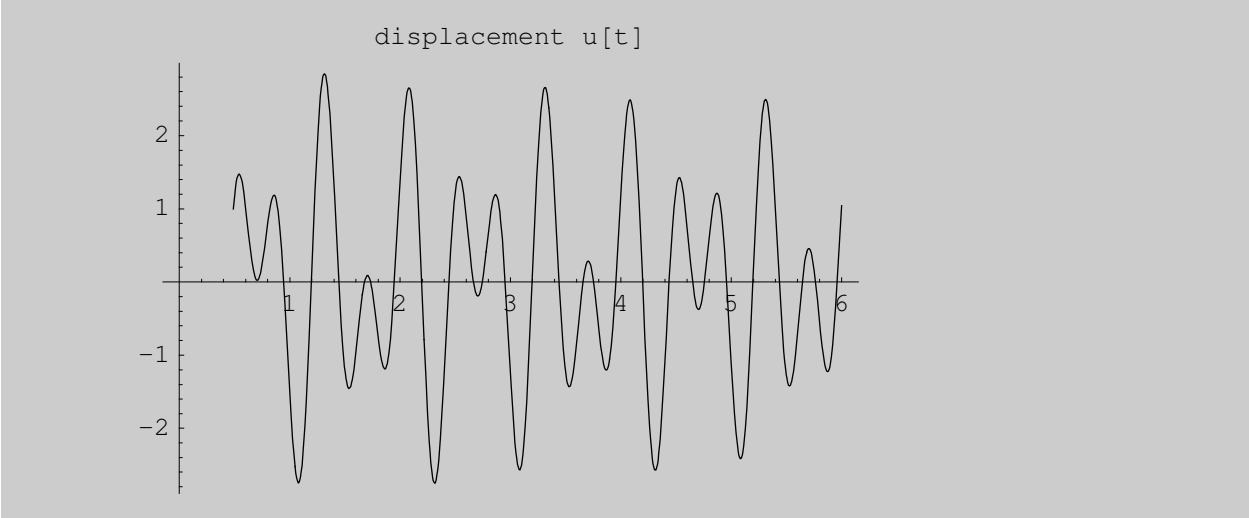
■ Numerical Approximation of Solution of the system of differential equations

We may not have been able to obtain an analytic solution, but we can solve numerically using NDSolve:

```
Clear[u, z]
solution2 =
 Flatten[NDSolve[{z'[t] + 2 alpha beta xi z[t] + alpha^2 omega^2 u[t] == -ug[t],
    u'[t] == z[t], z[t0] == yprime0, u[t0] == y0}, {z[t], u[t]}, {t, t0, 6}]]
```

```
{z[t] → InterpolatingFunction[{{0.48843, 6.}}, <>][t],
 u[t] → InterpolatingFunction[{{0.48843, 6.}}, <>][t]}
```

```
plot2 = Plot[Evaluate[u[t] /. solution2],
  {t, t0, tend}, PlotLabel → "displacement u[t]"]
```



□ Comparison of ug[t] from function and from input data

```
Show[plot1, plot2, PlotLabel → "Displacement from function and data file"]
```

■ What If

The input data for ug[t] did not give an accurate result because there were too few points to adequately describe the forcing function. Try solving the problem again with the file *quakedata2*, which contains the following points:

```
0.48843     236.047
0.964009    128.571
1.43959     −139.806
1.91517     −233.222
2.39074     −34.7715
2.86632     207.194
3.3419      189.866
3.81748     −65.0713
4.29306     −238.575
4.76864     −113.512
5.24421     153.605
5.71979     228.493
6.19537     17.4318
6.67095     −215.444
7.14653     −178.702
7.62211     81.6783
8.09769     239.841
8.57326     97.8538
9.04884     −166.594
9.52442     −222.557
                   −15
10.     −1.35308 10
```

A regression approximation of these data in the form `a Sin[δ t]` yields 240.00 Sin[15.7079 t]. This is a

good approximation which will yield a good result.

Solution of 2-D Laplace Equation

Introduction

The two-dimensional Laplace Equation

$$\frac{\partial^2 \theta}{\partial x^2} + \frac{\partial^2 \theta}{\partial y^2} + \frac{1}{k} g(x, y) = 0$$

describes two-dimensional, steady state conduction heat transfer. The solution θ(x,y) gives the temperature T(x,y)- Tinf on a rectangular plate, where Tinf is the temperature of the surroundings. The physical domain is given as 0 <= x <= PlateLength, 0 <= y <= PlateHeight.

An energy source term g(x,y) is allowed. Admissable boundary conditions are Type 1. The user specifies the material thermal conductivity k; the boundary functions; the values of the plate dimensions PlateLength and PlateHeight; and the desired numbers of subintervals in each direction. The program then constructs the finite difference equation at each interior node and solves for the desired nodal temperature excesses via the Solve function. The results are presented in the form of contour and surface plots.
The program may be modified to handle three-dimensional geometry, time-dependence and boundary conditions of Types 2 and 3. Also note that Solve is used only for illustrative purposes on this small toy problem. In other documents, we have used more sophisticated matrix methods. Also note that a variant of the heat equation is the fundamental formula underlying many financial derivatives, and in other papers these techniques are applied to problems of financial analysis.

Example

Determine the temperatures on a rectangular plate. The plate is internally heated by an energy source term g(x,y) = 1000 Exp[-4 Pi^2 x y]. The plate is 1.0 m by 1.0 m in dimension with thermal conductivity k = 275 W/m-K. The boundary conditions are:

(1) T[x,y] = 100 Sin[Pi y/PlateHeight], x = 0, 0 <=y<=PlateHeight;

(2) T[x,y] = 400 Sin[Pi x/(2 PlateLength)], y = PlateHeight, 0 <=x<=PlateLength;

(3) T[x,y] = 400 y/ PlateHeight, x = PlateLength, 0 <=y<=PlateHeight;

(4) T[x,y] = 0, y = 0, 0 <=x<=PlateLength.

Five subintervals will be taken in each direction.

Solution Technique

The finite-difference methods will be used to solve for the steady temperatures on a square plate subject to Type 1 boundary conditions T(x,0) = T(0,y) = 100; T(x,H) = T(L,y) = 0. The plate is subdivided into four equally-spaced subdivisions in each direction. The plate thermal conductivity is uniform and there is no internal energy generation. The governing equation is

D[T[x,y],[x,2}] + D[T[x,y],{y,2}] == 0 which is written in finite difference form as $T_{i,j+1} + T_{i,j-1} + T_{i+1,j} + T_{i-1,j} - 4T_{i,j} = 0$,

i = 1, ... , 4, j = 1, ... , 4.

Solution

The finite-difference equation is defined as eq[i,j]:

```
Clear[u, g, eq, T]
PlateLength = 1.0;  (* m *)
PlateHeight = 1.0;  (* m *)
NumberofXSubintervals = 5;
NumberofYSubintervals = 5;
DeltaX = PlateLength / NumberofXSubintervals;
DeltaY = PlateHeight / NumberofYSubintervals;
k = 275; (* W/m-K *)
g[i_, j_] = 1000 Exp[N[-4 Pi^2 i DeltaX j DeltaY]];
equation[i_, j_] := ((u[i + 1, j] - 2 u[i, j] + u[i - 1, j]) / DeltaX^2 +
    (u[i, j + 1] - 2 u[i, j] + u[i, j - 1]) / DeltaY^2 == 0)
```

```
— General::spell1 :
   Possible spelling error: new symbol name "NumberofYSubintervals" is similar
     to existing symbol "NumberofXSubintervals".

— General::spell1 :
   Possible spelling error: new symbol name "DeltaY" is similar to existing symbol "DeltaX".
```

Boundary Conditions

```
(* Along the side x = 0, 0 ≤ y ≤ PlateHeight *)
u[0, j_] := N[100 Sin[Pi j / NumberofYSubintervals] ]

(* Along the side y=PlateHeight,0<=x<=PlateLength *)
u[i_, NumberofYSubintervals] := 400 Sin[Pi i / (2 NumberofXSubintervals)]

(*  Along the side x=PlateLength,0<=y<=PlateHeight *)
u[NumberofXSubintervals, j_] := 400 j / NumberofYSubintervals
(* along the side y=0,0<=x<=PlateLength *)
u[i_, 0] :=  0
```

Now construct and print out the finite difference equations to be solved (one equation is written for each node at which the temperature is unknown.:

Mathematica constructs the list of equations, one at each point at which the temperature is unknown, with the following command:

```
Equations = Flatten[
              Table[
               Simplify [ equation[i, j] ] ,
   {i, NumberofXSubintervals - 1}, {j, NumberofYSubintervals - 1}]
 ]
```

```
{1469.46 + 25. u[1, 2] + 25. u[2, 1] == 100. u[1, 1],
 2377.64 + 25. u[1, 1] + 25. u[1, 3] + 25. u[2, 2] == 100. u[1, 2],
 2377.64 + 25. u[1, 2] + 25. u[1, 4] + 25. u[2, 3] == 100. u[1, 3],
 4559.63 + 25. u[1, 3] + 25. u[2, 4] == 100. u[1, 4],
 25. (u[1, 1] + u[2, 2] + u[3, 1]) == 100. u[2, 1],
 25. (u[1, 2] + u[2, 1] + u[2, 3] + u[3, 2]) == 100. u[2, 2],
 25. (u[1, 3] + u[2, 2] + u[2, 4] + u[3, 3]) == 100. u[2, 3],
 5877.85 + 25. u[1, 4] + 25. u[2, 3] + 25. u[3, 4] == 100. u[2, 4],
 25. (u[2, 1] + u[3, 2] + u[4, 1]) == 100. u[3, 1],
 25. (u[2, 2] + u[3, 1] + u[3, 3] + u[4, 2]) == 100. u[3, 2],
 25. (u[2, 3] + u[3, 2] + u[3, 4] + u[4, 3]) == 100. u[3, 3],
 8090.17 + 25. u[2, 4] + 25. u[3, 3] + 25. u[4, 4] == 100. u[3, 4],
 2000. + 25. u[3, 1] + 25. u[4, 2] == 100. u[4, 1],
 4000. + 25. u[3, 2] + 25. u[4, 1] + 25. u[4, 3] == 100. u[4, 2],
 6000. + 25. u[3, 3] + 25. u[4, 2] + 25. u[4, 4] == 100. u[4, 3],
 17 510.6 + 25. u[3, 4] + 25. u[4, 3] == 100. u[4, 4]}
```

Now write the unknowns

```
Unknowns = Flatten[ Table[u[i, j],
   {i, NumberofXSubintervals - 1}, {j, NumberofYSubintervals - 1}]]
```

```
{u[1, 1], u[1, 2], u[1, 3], u[1, 4], u[2, 1], u[2, 2], u[2, 3], u[2, 4],
 u[3, 1], u[3, 2], u[3, 3], u[3, 4], u[4, 1], u[4, 2], u[4, 3], u[4, 4]}
```

Show a schematic of the plate nodal temperatures:

```
MatrixForm[ Temperatures  =  Table[u[i, j],
   {i, 0, NumberofXSubintervals},  {j, 0, NumberofYSubintervals} ] ]
```

$$\begin{pmatrix} 0. & 58.7785 & 95.1057 & 95.1057 & 58.7785 & 0. \\ 0 & u[1,1] & u[1,2] & u[1,3] & u[1,4] & 100\left(-1+\sqrt{5}\right) \\ 0 & u[2,1] & u[2,2] & u[2,3] & u[2,4] & 100\sqrt{2\left(5-\sqrt{5}\right)} \\ 0 & u[3,1] & u[3,2] & u[3,3] & u[3,4] & 100\left(1+\sqrt{5}\right) \\ 0 & u[4,1] & u[4,2] & u[4,3] & u[4,4] & 100\sqrt{2\left(5+\sqrt{5}\right)} \\ 0 & 80 & 160 & 240 & 320 & 400 \end{pmatrix}$$

Now solve the system of linear equations for the unknowns:

```
result = Solve[Equations, Unknowns]
```

```
{{u[1, 1] → 50.8159, u[1, 2] → 90.5593, u[1, 3] → 113.124, u[1, 4] → 120.593,
  u[2, 1] → 53.9256, u[2, 2] → 103.192, u[2, 3] → 146.236, u[2, 4] → 186.863,
  u[3, 1] → 61.6947, u[3, 2] → 122.046, u[3, 3] → 181.767, u[3, 4] → 245.51,
  u[4, 1] → 70.8067, u[4, 2] → 141.532, u[4, 3] → 213.275, u[4, 4] → 289.802}}
```

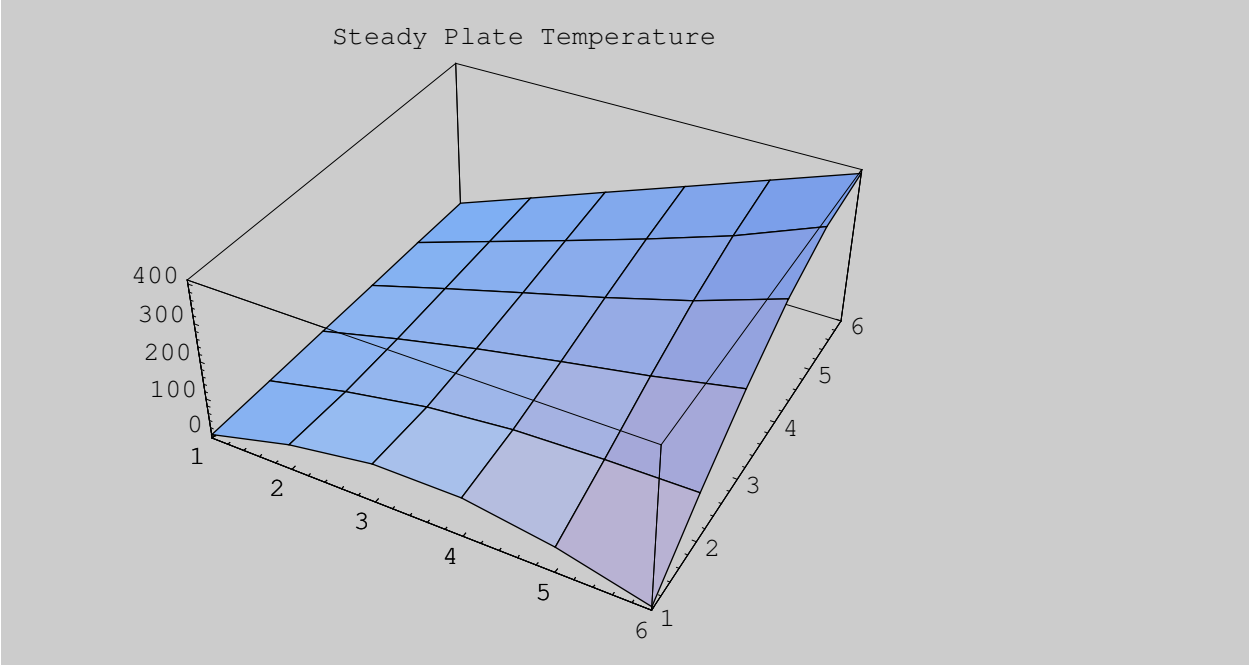
Put the temperatures in a convenient form for plotting:

```
Temperatures = Temperatures /. Flatten[result]
```

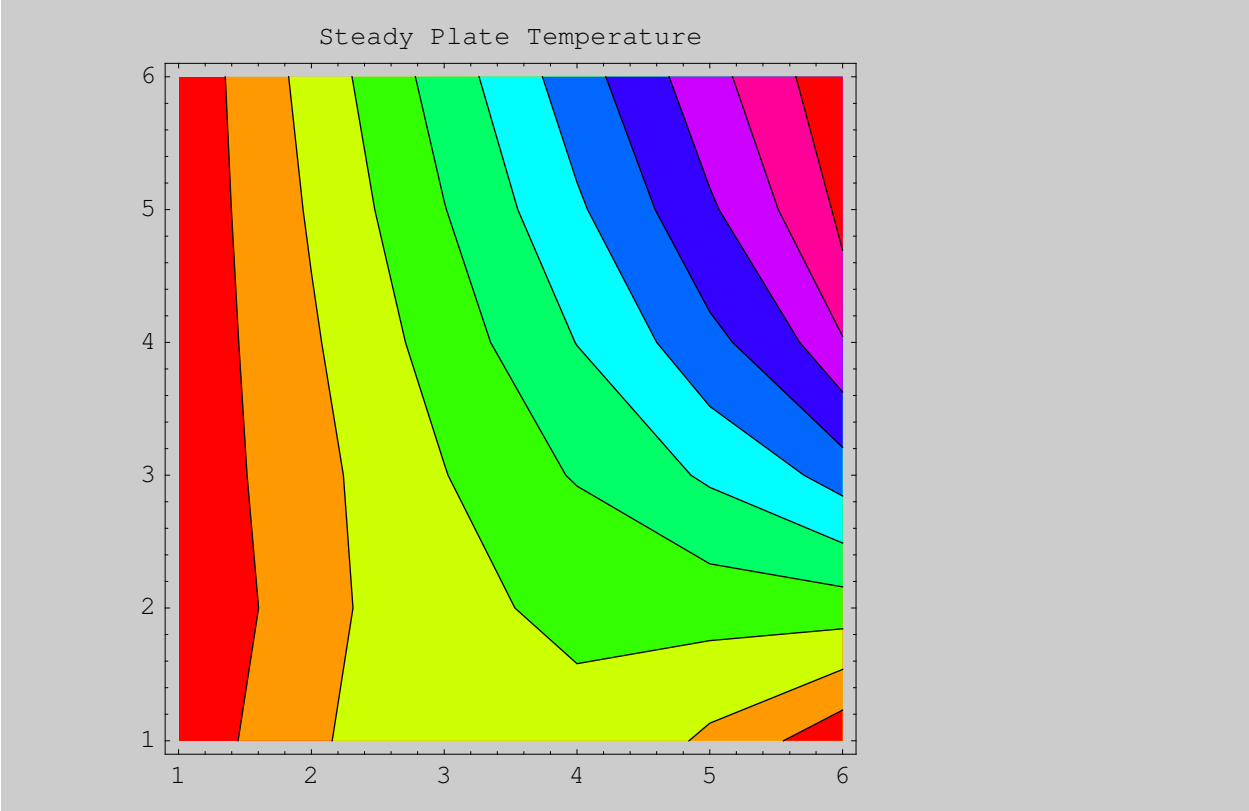
$$\left\{\{0., 58.7785, 95.1057, 95.1057, 58.7785, 0.\},\ \left\{0, 50.8159, 90.5593, 113.124, 120.593, 100\left(-1+\sqrt{5}\right)\right\},\ \left\{0, 53.9256, 103.192, 146.236, 186.863, 100\sqrt{2\left(5-\sqrt{5}\right)}\right\},\ \left\{0, 61.6947, 122.046, 181.767, 245.51, 100\left(1+\sqrt{5}\right)\right\},\ \left\{0, 70.8067, 141.532, 213.275, 289.802, 100\sqrt{2\left(5+\sqrt{5}\right)}\right\},\ \{0, 80, 160, 240, 320, 400\}\right\}$$

Plots

```
ListPlot3D[Temperatures, PlotLabel → "Steady Plate Temperature"]
```




```
ListContourPlot[Temperatures,
 PlotLabel → "Steady Plate Temperature", ColorFunction → Hue]
```


Steady Plate Temperature
6
5
4
3
2
1
1
2
3
4
5
6

Appendix

Code for CubicSplineInterpolation.m

```
BeginPackage["NumericalMath`SplineInterpolation`"]
SplineInterpolation::"usage" =
 "The package SplineInterpolation contains code for  construction of
   natural, clamped, periodic and B cubic splines. A plotting utility,
   PiecewiseCubicPlot, and integration procedure, IntegrateSpline,
   are also provided.  For information on usage of these procedures,
   type ?NaturalCubicSpline, ?ClampedCubicSpline, ?PeriodicSpline,
   ?BSplineInterpolation, ?PiecewiseCubicPlot or ?IntegrateSpline. "
NaturalCubicSpline::"usage" =
 "NaturalCubicSpline[{{x0,y0},{x1,y1},...{xn,yn}}]
   constructs and prints a natural cubic spline
   interpolating the given   data points. The natural spline
   boundary conditions   S''[x0] = S''[xn]=0 are used."
ClampedCubicSpline::"usage" =
 "ClampedCubicSpline[{{x0,y0},{x1,y1},...{xn,yn},{y'0,y'n}}]
   constructs and prints a clamped cubic spline interpolating
   the given   data points. The clamped spline boundary
   conditions   S'[x0] = f'[x0],S'[xn] = f'[xn] are used."
BSplineInterpolation::"usage" =
 "BSplineInterpolation[{{x0,y0},{x1,y1},...{xn,yn},{y'0,y'n}}]
   constructs and prints a B-spline interpolating the given
   data points. Both natural and clamped spline boundary
   conditions S'[x0] = f'[x0]; S'[xn] = f'[xn];   S''[x0] =
   S''[xn] = 0 are used.  The B-Spline does   not interpolate the
   given data at the points   {{x1,f[x1]},{xn-1,f[xn-1]}}.
   NOTE:  n must be >2 for BSplineInterpolation."
PeriodicSpline::"usage" = "PeriodicSpline[{{x0,y0},{x1,y1},...{xn,yn}}]
   constructs and plots a periodic spline.  This procedure is used for
   plotting smooth, closed curves.  The periodic boundary conditions
   S[x0] = S[xn], S'[x0] = S'[xn] and S''[x0] = S''[xn] are used."
IntegrateSpline::"usage" = "IntegrateSpline computes the
   integral of the spline     over the interval [x[0],x[n]].
   It is not intended for use     with PeriodicSpline."
CubicSpline::"smallnerr" = "Number of data points, n =`1` is  insufficient
   for spline interpolation: you must provide at least `2` data points."
CubicSpline::"nonfunctionerr" = "The x coordinates provided
   `1` do not describe a function.  The x's must be unique."
PiecewiseCubicPlot::"usage" = "PiecewiseCubicPlot[]    constructs a plot
   of the constructed spline.  The option    PlotPoints->num may be
   included.  Otherwise, the default    value of PlotPoints is used.
```

```
   Note that setting num=1 yields    a piecewise linear plot, identical
   to the result    obtained via ListPlot, with PlotJoined->True.
   NOTE:  Either NaturalCubicSpline or ClampedCubicSpline
   must be executed prior to calling PiecewiseCubicPlot."
Begin["`Private`"]
Unprotect[ClampedCubicSpline, NaturalCubicSpline, IntegrateSpline,
 PiecewiseCubicPlot, BSplineInterpolation, S, PeriodicSpline]
S[i_, x_] := a[i] + b[i] (x - t[i]) + c[i] (x - t[i])^2 + d[i] (x - t[i])^3
WriteEquations := Block[{i}, EquationList = {};
  EquationList = Append[EquationList, S[n - 1, t[n]] == a[n]];
  Do[EquationList = Append[EquationList, S[i, t[i + 1]] == S[i + 1, t[i + 1]]];
   EquationList = Append[EquationList,
     Expand[∂x S[i, x] /. x → t[i + 1]] == Expand[∂x S[i + 1, x] /. x → t[i + 1]]];
   EquationList = Append[EquationList, Expand[∂{x,2} S[i, x] /. x → t[i + 1]] ==
      Expand[∂{x,2} S[i + 1, x] /. x → t[i + 1]]], {i, 0, n - 2}]]
MakeUnknowns := Block[{i}, Unknowns = {};
  Do[Unknowns = Append[Unknowns, b[i]], {i, 0, n - 1}];
  Do[Unknowns = Append[Unknowns, c[i]], {i, 0, n - 1}];
  Do[Unknowns = Append[Unknowns, d[i]], {i, 0, n - 1}]]
SolvetheSystem := Block[{}, Result = N[Solve[EquationList, Unknowns]];
  Do[b[i - 1] = Result[[1, i, 2]]; c[i - 1] = Result[[1, i + n, 2]];
   d[i - 1] = Result[[1, i + 2 n, 2]], {i, 1, 3 n/3}];
  Print["The spline is constructed as follows: "];
  Do[Print["S[", i, ",x] = ", Expand[S[i, x]],
    "  for ", t[i], " < x < ", t[i + 1]], {i, 0, n - 1}]]
SolvetheBSplineSystem := Block[{}, Result = N[Solve[EquationList, Unknowns]];
  a[1] = Result[[1, 1, 2]]; a[n - 1] = Result[[1, 2, 2]];
  Do[b[i - 3] = Result[[1, i, 2]]; c[i - 3] = Result[[1, i + n, 2]];
   d[i - 3] = Result[[1, i + 2 n, 2]], {i, 3, 3 + n - 1}];
  Print["The B spline is constructed as follows: "];
  Do[Print["S[", i, ",x] = ", Expand[S[i, x]],
    "  for ", t[i], " < x < ", t[i + 1]], {i, 0, n - 1}]]
SolvePeriodicSystem := Block[{}, LHS = Table[Row[i], {i, 0, 3 n - 1}];
  RHS = Table[constant[i], {i, 0, 3 n - 1}];
  Result = LinearSolve[LHS, RHS]; Do[b[i - 1] = Result[[i]];
   c[i - 1] = Result[[i + n]]; d[i - 1] = Result[[i + 2 n]], {i, 1, n}];]
WriteandSolvePeriodicEquations := Block[{i}, Clear[Row, Equation, b, c, d];
  Do[Equation[i] = (S[i, t[i + 1]] - a[i]) - (S[i + 1, t[i + 1]] - a[i + 1]);
   constant[i] = -a[i] + a[i + 1]; Equation[i + n - 1] =
    Expand[∂x S[i, x] /. x → t[i + 1]] - Expand[∂x S[i + 1, x] /. x → t[i + 1]];
   constant[i + n - 1] = 0; Equation[i + 2 n - 2] =
    Expand[∂{x,2} S[i, x] /. x → t[i + 1]] - Expand[∂{x,2} S[i + 1, x] /. x → t[i + 1]];
```

```
    constant[i + 2 n - 2] = 0, {i, 0, n - 2}];
   Equation[3 n - 3] = S[n - 1, t[n]] - a[n - 1] - S[0, t[0]];
   constant[3 n - 3] = a[0] - a[n - 1]; Equation[3 n - 2] =
    Expand[∂x S[0, x] /. x → t[0]] - Expand[∂x S[n - 1, x] /. x → t[n]];
   constant[3 n - 2] = 0; Equation[3 n - 1] =
    Expand[∂{x,2} S[0, x] /. x → t[0]] - Expand[∂{x,2} S[n - 1, x] /. x → t[n]];
   constant[3 n - 1] = 0; Do[Row[j] = {}; Row[j] = Append[Row[j],
      Table[Coefficient[Equation[j], b[i]], {i, 0, n - 1}]]; Row[j] = Append[
      Row[j], Table[Coefficient[Equation[j], c[i]], {i, 0, n - 1}]]; Row[j] =
     Append[Row[j], Table[Coefficient[Equation[j], d[i]], {i, 0, n - 1}]];
    Row[j] = Flatten[Row[j]], {j, 0, 3 n - 1}];
   MakeUnknowns; SolvePeriodicSystem]
NaturalCubicSpline[slist_List] :=
 Block[{i}, Clear[a, b, c, d]; SplineData = N[slist];
   MinNumberofPoints = 3; If[Length[SplineData] < MinNumberofPoints,
    Message[CubicSpline::"smallnerr", Length[SplineData],
     MinNumberofPoints]; Return[Hold[NaturalCubicSpline[data]]]];
   If[SplineData ≠ Sort[SplineData], SplineData = Sort[SplineData]];
   xlist = Table[SplineData[[i, 1]], {i, Length[SplineData]}];
   If[xlist ≠ Union[xlist], Message[CubicSpline::"nonfunctionerr", xlist];
    Return[]]; n = Length[SplineData] - 1; Do[t[i] = SplineData[[i + 1, 1]];
    a[i] = SplineData[[i + 1, 2]], {i, 0, n}]; WriteEquations;
   EquationList = Append[EquationList, Expand[∂{x,2} S[0, x] /. x → t[0]] == 0];
   EquationList = Append[EquationList, Expand[∂{x,2} S[n - 1, x] /. x → t[n]] == 0];
   MakeUnknowns; SolvetheSystem]
ClampedCubicSpline[slist_List] :=
 Block[{i}, Clear[a, b, c, d]; SplineData = N[slist];
   MinNumberofPoints = 4; If[Length[SplineData] < MinNumberofPoints,
    Message[CubicSpline::"smallnerr", Length[SplineData],
     MinNumberofPoints]; Return[Hold[ClampedCubicSpline[data]]]];
   LeftDeriv = SplineData[[Length[SplineData], 1]];
   RightDeriv = SplineData[[Length[SplineData], 2]];
   SplineData = Drop[SplineData, -1];
   If[SplineData ≠ Sort[SplineData], SplineData = Sort[SplineData]];
   xlist = Table[SplineData[[i, 1]], {i, Length[SplineData]}];
   If[xlist ≠ Union[xlist], Message[CubicSpline::"nonfunctionerr", xlist];
    Return[]]; n = Length[SplineData] - 1;
   Do[t[i] = SplineData[[i + 1, 1]]; a[i] = SplineData[[i + 1, 2]], {i, 0, n}];
   WriteEquations; EquationList = Append[EquationList,
     Expand[∂x S[0, x] /. x → t[0]] == LeftDeriv]; EquationList =
    Append[EquationList, Expand[∂x S[n - 1, x] /. x → t[n]] == RightDeriv];
   MakeUnknowns; SolvetheSystem]
BSplineInterpolation[slist_List] :=
 Block[{i}, Clear[a, b, c, d]; SplineData = N[slist];
   MinNumberofPoints = 5; If[Length[SplineData] < MinNumberofPoints,
```

```
    Message[CubicSpline::"smallnerr", Length[SplineData],
     MinNumberofPoints]; Return[Hold[BSplineInterpolation[data]]]];
   LeftDeriv = SplineData[[Length[SplineData], 1]];
   RightDeriv = SplineData[[Length[SplineData], 2]];
   SplineData = Drop[SplineData, -1];
   If[SplineData ≠ Sort[SplineData], SplineData = Sort[SplineData]];
   xlist = Table[SplineData[[i, 1]], {i, Length[SplineData]}];
   If[xlist ≠ Union[xlist],
    Message[CubicSpline::"nonfunctionerr", xlist]; Return[]];
   n = Length[SplineData] - 1; Do[t[i] = SplineData[[i + 1, 1]], {i, 0, n}];
   a[0] = SplineData[[1, 2]]; a[n] = SplineData[[n + 1, 2]];
   Do[a[i] = SplineData[[i + 1, 2]], {i, 2, n - 2}]; WriteEquations;
   EquationList = Append[EquationList, Expand[∂{x,2} S[0, x] /. x → t[0]] == 0];
   EquationList = Append[EquationList, Expand[∂{x,2} S[n - 1, x] /. x → t[n]] == 0];
   EquationList = Append[EquationList,
     Expand[∂x S[0, x] /. x → t[0]] == LeftDeriv]; EquationList =
    Append[EquationList, Expand[∂x S[n - 1, x] /. x → t[n]] == RightDeriv];
   MakeUnknowns; Unknowns = Prepend[Unknowns, a[n - 1]];
   Unknowns = Prepend[Unknowns, a[1]]; SolvetheBSplineSystem]
PeriodicSpline[slist_List, opts___Rule] :=
 Block[{i, plotpoints, X, Y}, Clear[a, b, c, d];
   plotpoints = PlotPoints /. {opts} /. Options[Plot]; SplineData = slist;
   MinNumberofPoints = 3; If[Length[SplineData] < MinNumberofPoints,
    Message[CubicSpline::"smallnerr", Length[SplineData],
     MinNumberofPoints]; Return[Hold[NaturalCubicSpline[data]]]];
   n = Length[SplineData]; Do[t[i] = i + 1; a[i] = SplineData[[i + 1, 1]];
    aa[i] = SplineData[[i + 1, 2]], {i, 0, n - 1}]; t[n] = n + 1; a[n] = a[0];
   WriteandSolvePeriodicEquations; title = "PeriodicCubicSpline";
   Do[X[i] = Drop[Table[N[S[i, x]], {x, t[i], t[i + 1], (t[i + 1] - t[i])/plotpoints}], -1],
    {i, 0, n - 2}];
   X[n - 1] = Table[N[S[n - 1, x]], {x, t[n - 1], t[n], (t[n] - t[n - 1])/plotpoints}];
   AbscissaPoints = Flatten[Table[X[i], {i, 0, n - 1}], 1]; Clear[a, b, c, d];
   Do[a[i] = aa[i], {i, 0, n - 1}]; a[n] = a[0]; WriteandSolvePeriodicEquations;
   Do[Y[i] = Drop[Table[N[S[i, x]], {x, t[i], t[i + 1], (t[i + 1] - t[i])/plotpoints}], -1],
    {i, 0, n - 2}];
   Y[n - 1] = Table[N[S[n - 1, x]], {x, t[n - 1], t[n], (t[n] - t[n - 1])/plotpoints}];
   OrdinatePoints = Flatten[Table[Y[i], {i, 0, n - 1}], 1]; SplineData = Table[
     {AbscissaPoints[[i]], OrdinatePoints[[i]]}, {i, Length[OrdinatePoints]}];
   ListPlot[SplineData, Joined → True, PlotLabel → title]]
```

```
PiecewiseCubicPlot[opts___Rule] := Block[{i, plotpoints},
  plotpoints = PlotPoints /. {opts} /. Options[Plot];
  title = PlotLabel /. {opts} /. Options[Plot];
  Do[plot[i] = Table[N[{x, S[i, x]}], {x, t[i], t[i + 1], (t[i + 1] - t[i])/plotpoints}],
   {i, 0, n - 1}]; SplinePlot = ListPlot[Flatten[
     Table[plot[i], {i, 0, n - 1}], 1], Joined → True, PlotLabel → title]]
IntegrateSpline := Block[{i}, SplineIntegral =
   Sum[a[i] (t[i + 1] - t[i]) + 1/2 b[i] (t[i + 1] - t[i])^2 +
       1/3 c[i] (t[i + 1] - t[i])^3 + 1/4 d[i] (t[i + 1] - t[i])^4, {i, 0, n - 1}];
  Print["The integral of the cubic spline is ", SplineIntegral, "."]]
End[]
Protect[NaturalCubicSpline, ClampedCubicSpline, BSplineInterpolation,
 PiecewiseCubicPlot, PeriodicSpline, IntegrateSpline]
EndPackage[]
```