Report from the Trenches: Tips, Tricks and Traps for *Mathematica* Programmers

Jennifer Voitle
jennifer@treasuryfinance.com

Introduction

A mindmap of some of the items discussed in this presentation follows. (The mindmap was brainstormed in Inspiration 4.0 and imported into *Mathematica*.) (mindmap disappeared, sorry!)

Why Program in *Mathematica*?

```
Off[General::"spell1"]
Off[General::"spell"]
```

- **Notebooks!**

- **Variety of Programming Styles!**

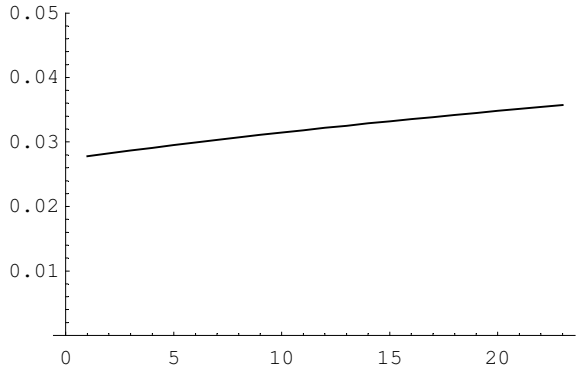
```
newtemps  =   Drop[Temperatures,-1] +
    Deltat dTdt;
Temperatures =
    AppendTo[newtemps,newtemps[[Nd2]] ];
```

Applications

Air Properties

Air Property data are in form {T [F], k [BTU/hr-ft-F],rho in {lb/ft^3], mu in [lbm/ft-hr] }

```
Temperature = Flatten[Join[Table[100 + 20 i, {i, 0, 45}], Table[1050 + 50 i, {i, 0, 9}]]];
Conductivity =
  1/3600 {.01566, .01615, .01664, .01712, .01759, .01806, .01853, .01899, .01945, .0199,
    .02034, .02079, .02122, .02166, .02208, .02251, .02293, .02335, .02376, .02417,
    .02458, .02498, .02538, .02577, .02616, .02655, .02694, .02732, .0277,
    .02807, .02844, .02881, .02918, .02954, .0299, .03026, .03062, .03097,
    .03132, .03167, .03201, .03235, .03269, .03303, .03337, .0337, .03452,
    .03533, .03613, .03691, .03768, .03844, .03919, .03993, .04066, .04137};
rho = {.07087, .06843, .06614, .06401, .06201, .06013, .05836, .05669, .05512,
   .05363, .05221, .05087, .0496, .04839, .04724, .04614, .04509, .04409,
   .04313, .04221, .04133, .04049, .03968, .0389, .03815, .03743, .03673, .03607,
   .03543, .03481, .0342, .03362, .03306, .03252, .032, .03149, .031, .03052,
   .03006, .02961, .02917, .02875, .02834, .02794, .02755, .02717, .02627,
   .02543, .02464, .0239, .0232, .02254, .02192, .02133, .02077, .02024};
mu = 1/3600 {.04594, .04718, .04839, .04959, .05077, .05193, .05308, .0542, .05531,
    .0564, .05748, .05854, .05959, .06063, .06165, .06266, .06366, .06464, .06561,
    .06657, .06752, .06846, .06939, .07031, .07122, .07212, .07301, .07389,
    .07477, .07563, .07649, .07734, .07818, .07901, .07984, .08066, .08147,
    .08227, .08307, .08386, .08464, .08542, .0862, .08696, .08772, .08847, .09034,
    .09216, .09396, .09572, .09746, .09917, .10085, .1025, .10414, .10575};
NPr = {.706, .703, .7, .698, .696, .694, .693, .691, .689, .688, .687, .686, .685,
   .684, .683, .682, .682, .681, .681, .68, .68, .68, .68, .681, .681, .681, .681, .682,
   .682, .682, .683, .683, .683, .684, .684, .685, .686, .686, .686, .687, .688, .689,
   .69, .69, .69, .691, .693, .695, .697, .698, .7, .702, .703, .705, .707, .709};
ThermalConductivity = Transpose[{Temperature, Conductivity}];
AirDensity = Transpose[{Temperature, rho}];
DynamicViscosity = Transpose[{Temperature, mu}];
PrandtlNumber = Transpose[{Temperature, NPr}];
k = Interpolation[ThermalConductivity];
viscosity = Interpolation[DynamicViscosity];
Pr = Interpolation[PrandtlNumber];
density = Interpolation[AirDensity];
Clear[h, gn]
Attributes[h] = {Listable};
NeededTemperatures = Table[50 i, {i, 2, 24}];
Rv = 1; Rm = 1; c = 0.85; Deltap = 80; Nn = .314;
gn := 15.56 Sqrt[density[100] Deltap];
FilmTemperatures = (NeededTemperatures + 100)/2;
h[T_] := (.815 Rv^.625 Nn^.188 Pr[T]^(1/3) k[T] gn^.625)/viscosity[T]^.625;
ListPlot[h[FilmTemperatures], PlotJoined → True, PlotRange → {0, .05}];
```



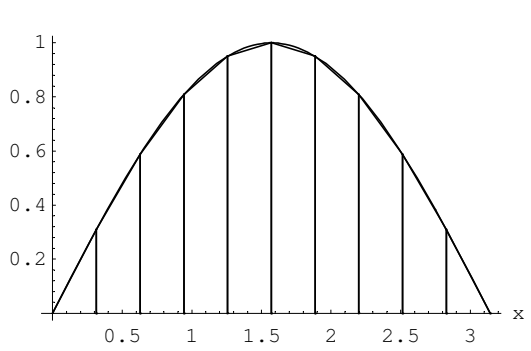
■ Application: Integration by the Trapezoid Rule

To perform numerical integration, the intrinsic function NIntegrate may be used. However, the trapezoid method is illustrated as it is a good example of the improvement that can be made by using functional rather than procedural programming in *Mathematica*.

The basic idea is: given an integrand over the interval {a,b}, partition into n uniformly spaced subintervals of size h = (b-a)/n. The function is evaluated at each point a + i h, and the integral computed as the sum of the integrals over all subintervals. Thus, the scheme is:

(Later, we shall treat non-uniformly spaced data.) We shall apply the rule to the sine function on $\{0,\pi\}$, using 10 subintervals. A plot of the composite trapezoid rule can be generated as follows:

```
sinPlot = Plot[Sin[x], {x, 0, π}, AxesLabel → {"x", "Sin[x]"}, DisplayFunction → Identity];
points = Partition[Flatten[Table[
     {{i, 0}, {i, Sin[i]}, {i + π/10, Sin[i + π/10]}, {i + π/10, 0}}, {i, 0, 9 π/10, π/10}]], 2];
trapPlot = Show[Graphics[Line /@ Table[Take[points, {i, i + 1}],
      {i, 1, Length[points] - 1}]], DisplayFunction → Identity];
Show[sinPlot, trapPlot, DisplayFunction → $DisplayFunction];
```



■ FORTRAN Program

```
      f(x) = sin(x)
      a = 0
      b = 3.14159
      sum = 0.
      n = 10
      h = (b-a)/n
      Do 10 i = 1, n-1
         sum = sum + 2*f(a + i*h)
10    continue
      sum = sum + f(a) + f(b)
      sum = h/2*sum
      print *,'Integral approximated as: ',sum
      end
```

■ Procedural Approach in *Mathematica*

The above code can be duplicated in *Mathematica* via constructs such as For, Do, Table, and the like. We show Do here for direct comparison with Fortran.

```
f[x_] = Sin[x];
a = 0;
b = π;
sum = 0;
n = 10;
h = (b - a)/n;
Do[sum = sum + 2 f[a + i h], {i, 1, n - 1}];
sum = sum + f[a] + f[b];
sum = h sum/2;
Print["Integral approximated as: ", N[sum]]
```

```
Integral approximated as: 1.98352
```

The main difference between FORTRAN and *Mathematica* here is the lack of an End statement and Do label in the *Mathematica* code. Also, we were required to wrap N[] around the final result to obtain a numerical value. Unlike FORTRAN, *Mathematica* does not have default variable types. All of the above are assumed to be real and numeric.

```
Print["Integration by Composite Trapezoidal Rule"]
n = Input["Enter the number of data points (MAX 100)"]
Print["Enter the data points, one per line: "]
Do[{x[i], f[i]} = Input["{x,f[x]} = "], {i, 1, n}]
sum = 0;
Do[sum = sum + 1/2 (x[i + 1] - x[i]) (f[i + 1] + f[i]), {i, 1, n - 1}]
Print["Integral approximated as: ", sum]
```

■ Functional Approach in *Mathematica*

The preceding calculation can be substantially improved if we take advantage of *Mathematica*'s list structure. Actually, all that is really involved in the trapezoid rule does is creating a list of function values, summing them and multiplying by the appropriate weighting factor: h if an interior point, h/2 if an end point. Then let us attack the problem from this viewpoint. First, create the list of function values using Table:

```
f[x_] = Sin[x];
a = 0;
b = π;
n = 10;
h = (b - a)/n;
FunctionValues = Table[f[i], {i, a, b, h}]
```

$$\left\{0, \text{Sin}\left[\frac{\pi}{10}\right], \text{Sin}\left[\frac{\pi}{5}\right], \text{Sin}\left[\frac{3\pi}{10}\right], \text{Sin}\left[\frac{2\pi}{5}\right], 1, \text{Sin}\left[\frac{3\pi}{5}\right], \text{Sin}\left[\frac{7\pi}{10}\right], \text{Sin}\left[\frac{4\pi}{5}\right], \text{Sin}\left[\frac{9\pi}{10}\right], 0\right\}$$

Now, we'll sum them using the appropriate weighting factors. One way to accomplish this is to create a table of weighting factors {1,2,2,...2,1} and use the vector product to multiply.

```
Table[2, {i, 1, Length[FunctionValues]}] /. #1[[1]] → 1
```

```
{2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
```

```
N[1/2 h Plus @@ (%97.FunctionValues)]
```

```
1.983523537509454504
```

```
Take[FunctionValues, {2, -2}] /. {x__, y__} → x + y
```

$$1 + \text{Sin}\left[\frac{\pi}{10}\right] + \text{Sin}\left[\frac{\pi}{5}\right] + \text{Sin}\left[\frac{3\pi}{10}\right] + \text{Sin}\left[\frac{2\pi}{5}\right] + \text{Sin}\left[\frac{3\pi}{5}\right] + \text{Sin}\left[\frac{7\pi}{10}\right] + \text{Sin}\left[\frac{4\pi}{5}\right] + \text{Sin}\left[\frac{9\pi}{10}\right]$$

The above statements may be put together into a procedure which takes as input the list of function values and returns the computed sum.

```
TrapezoidRule[f_Symbol, {a_, b_}, n_Integer?Positive] :=
 Module[{h = (b - a)/n, FunctionValues = Table[f[i], {i, a, b, h}]},
  N[1/2 h (f[a] + f[b]) + Take[FunctionValues, {2, -2}] /. {x__, y__} → x + y]]
```

```
Timing[TrapezoidRule[Sin, {0, π}, 10]]
```

```
{2.3999999999999911 Second, 1.983523537509454504}
```

■ Thermodynamics

```
Needs["Engineering`Thermodynamics`"] (* my package, not part of Mathematica *)
interpolate[PressureTable, P == 0.2]
```

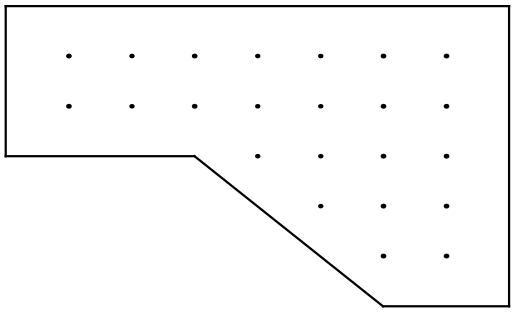
$$\left\{P \to 0.2\,\text{Mega Pascal},\ T \to 120.23\,\text{Celsius},\ vf \to \frac{0.001061\,m^3}{kg},\ vfg \to \frac{0.884639\,m^3}{kg},\right.$$
$$vg \to \frac{0.8857\,m^3}{kg},\ uf \to \frac{504.49\,kJ}{kg},\ ufg \to \frac{2025.\,kJ}{kg},\ ug \to \frac{2529.5\,kJ}{kg},\ hf \to \frac{504.7\,kJ}{kg},$$
$$\left.hfg \to \frac{2201.9\,kJ}{kg},\ hg \to \frac{2706.7\,kJ}{kg},\ sf \to \frac{1.5301\,kJ}{K\,kg},\ sfg \to \frac{5.597\,kJ}{K\,kg},\ sg \to \frac{7.1271\,kJ}{K\,kg}\right\}$$

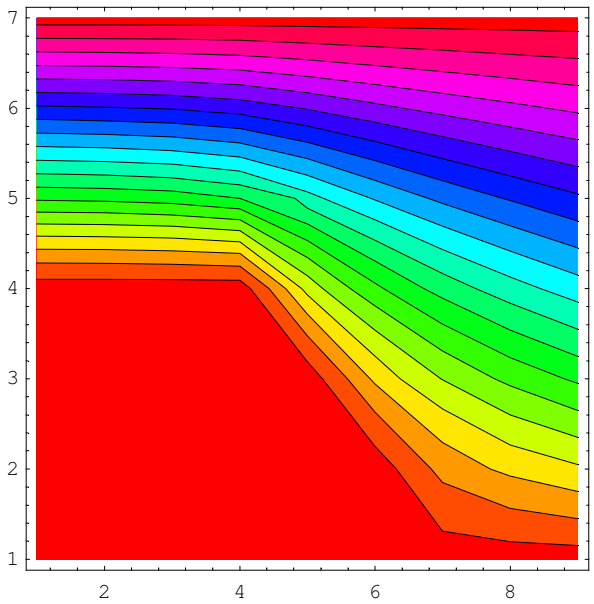
```
u[0.8]
```

```
404.4777
```

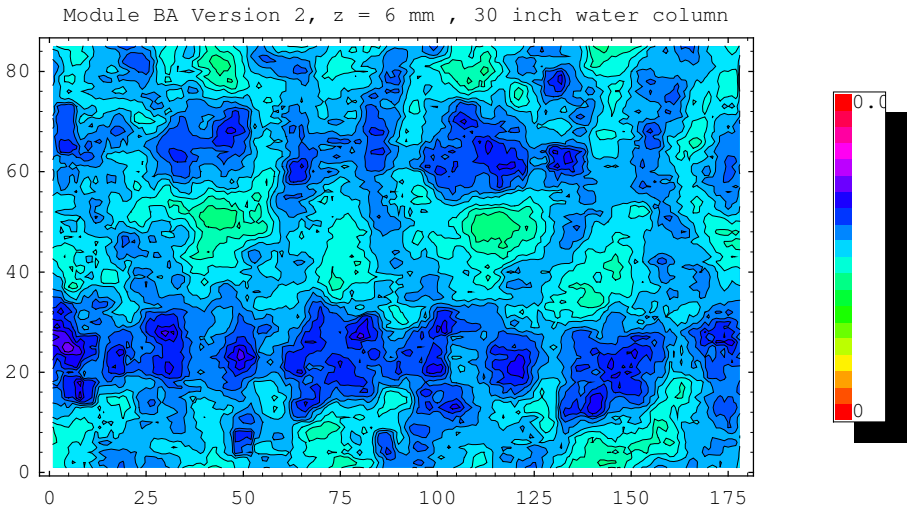
I can solve for any conditions, interpolating at constant entropy between two points, for example.

■ Flow Over a Conduit





■ Experimental Results



■ Versatility

Tips

■ Lists

Lists are very useful in *Mathematica*. Functions such as Transpose, . (vector dot product) are very useful. Often it is necessary to set the Attributes of a list to get the desired behavior.

■ Transpose

This is commonly used to combine two one-dimensional lists to create ordered pairs:

```
Transpose[{Range[10], Range[10]^3}]
```

```
{{1, 1}, {2, 8}, {3, 27}, {4, 64}, {5, 125},
 {6, 216}, {7, 343}, {8, 512}, {9, 729}, {10, 1000}}
```

An application where Transpose is useful is in building up property data tables, such as {T,k[T]}, {T, cp[T]}, etc. where you don't want to keep typing T.

■ Vector Dot Product

From the 1D Glass Tempering program, air properties are computed as

■ Coefficients for Thermophysical Properties of Gases

```
air[specificheat] = {0.239496, -3.89538/10^5, 1.3612/10^7, -6.4073/10^11};
air[density] = {35.989/10^3};
air[viscosity] = {2.2879728/10^6, 6.2597929/10^8, -3.1319564/10^11, 8.1503801/10^15, 3};
air[conductivity] = {1.3003035/10^3, 9.3676581/10^5, -4.4424691/10^8, 2.317158/10^11, -6.5997572/10^15};
CO2[specificheat] = {0.144592, 2.58445/10^4, -1.31976/10^7, 2.42548/10^11};
CO2[density] = {1.2543264/10^2, 5.2174396 10^2, 4.1683104 10^3};
CO2[viscosity] = {-1.2514321/10^7, 5.5159602/10^8, -1.5943186/10^11};
CO2[conductivity] = {-2.2264193/10^3, 4.7567075/10^5, 6.2085195/10^8, -3.7732778/10^11};
N2[specificheat] = {0.200469, 1.10194/10^4, -3.37414/10^8, 2.21178/10^12};
N2[density] = {6.8243029/10^3, 3.347189 10^2, 1.2159739 10^3};
N2[viscosity] = {-1.4524606/10^7, 7.7017551/10^8, -6.9103147/10^11, 4.4701282/10^14, -1.2082496/10^17};
N2[conductivity] = {7.7524326/10^6, 1.0136155/10^4, -5.733186/10^8, 1.8781222/10^11};
O2[specificheat] = {0.160484, 1.96817/10^4, -1.28451/10^7, 3.17949/10^11};
O2[density] = {5.237662/10^3, 3.8455445 10^2, 1.1669908 10^3};
O2[viscosity] = {2.9459735/10^7, 8.0358693/10^8, -4.6796539/10^11, 1.4476799/10^14};
O2[conductivity] = {-4.2081403/10^4, 1.0082565/10^4, -4.0254596/10^8, 1.1396071/10^11};
H2O[specificheat] = {0.443222, -2.85391/10^5, 1.98477/10^7, -6.63724/10^11};
H2O[density] = {9.1882182/10^3, 2.0661307 10^2, 4.6814874 10^3};
H2O[viscosity] = {-3.0769864/10^6, 4.0698422/10^8, -7.6276581/10^18};
H2O[conductivity] = {1.3046/10^2, -3.7561908/10^5, 2.2179639/10^7, -1.1115621/10^10};
Properties[prop_, T_] :=
 0.12 Plus @@ (CO2[prop].Table[T^i, {i, 0, Length[CO2[prop]] - 1}]) +
  0.18 Plus @@ (H2O[prop].Table[T^i, {i, 0, Length[H2O[prop]] - 1}]) +
  0.70 Plus @@ (N2[prop].Table[T^i, {i, 0, Length[N2[prop]] - 1}])
```

■ The Call

```
Properties[viscosity, 600]
```

```
0.00002760371598860775512
```

■ SequenceForm

Application: Lagrange Multiplier technique for Optimization. Calls FindRoot which needs a List, not a List[List]] as returned by FindRoot. Program has to accept list of user-provided initial guesses and pass to FindRoot, has to be in Sequence. (Don't want user to have to type something like {x->1, y-> 5, z-> 10} since the program generates Lagrange Multiplier(s) which also need initial guesses.)

■ Optimization by Lagrange Multipliers

First, load in our package:

```
Needs["Engineering`Optimization`"]
```

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

```
LagrangeMultiplier[{constraint
   function},{objective
   functions},{args},{initialguesses}]
   constructs the system of linear equations
   by the Lagrange Multiplier technique.  The
   initial guesses include the list of
   unknowns plus guesses for the lagrange
   multipliers (the number of these is the sum
   of  the number of unknowns plus the number
   of constraint functions + 1)

EXAMPLE:  LagrangeMultiplier[
{900 + 1100 d^2.5 l + 320 d l},
{50 Pi d^2 l - 100},{d,l},{0.8,1.4,9}]
```

If you don't have the engineering package, the input to LagrangeMultiplier is :

```
Unprotect[LagrangeMultiplier]
Clear[LagrangeMultiplier]
grad[f_List,arg_List] := Flatten[Outer[D,f,arg]]

LagrangeMultiplier[f_List,obj_List,arg_List,initguess_List] :=
Module[{},
equations:=If[Length[obj]<2,
   Flatten[
      Table[Outer[D,f,arg]-
         lambda[i] Outer[D,{obj[[i]]},arg],{i,Length[obj]}]],
Flatten[
   Table[Outer[D,f,arg]-lambda[i] Outer[D,{obj[[i]]},arg]-
      lambda[i+1] Outer[D,{obj[[i+1]]},arg],{i,Length[obj]-1}]]];

equations = Table[equations[[i]]==0,{i,Length[arg]}];
equations = AppendTo[equations,Table[obj[[i]]==0,{i,Length[obj]}]];
equations = Flatten[equations];
unknowns= arg;
unknowns = Flatten[AppendTo[unknowns,Table[lambda[i],{i,Length[obj]}]]];
(* result = N[Solve[equations,unknowns]]; *)
result = FindRoot[equations,Evaluate[
   Apply[Sequence,
     Transpose[{unknowns,initguesses}] ]] ];
Print[result]
]
Protect[LagrangeMultiplier]

Clear[cost,objective]
cost[d_,l_] = 900 + 1100 d^2.5 l + 320 d l;
objective[d_,l_] = 50 Pi d^2 l - 100;
LagrangeMultiplier[{cost[d,l]},{objective[d,l]},{d,l},{1,1,1}]
```

```
{d -> 0.696934, l -> 1.31068, lambda[1] -> 8.7692}
```

How does this work? We passed an objective function and cost function to minimize. If we try to use **FindRoot**, it expects it's arguments to be in the form lhs == rhs, {x,x0},{y,y0},{z,z0}, ...]. But we have passed in a list of unknowns {x,y,z,...} and initial guesses {x0,y0,z0,...}. How can we get this in the form {x,x0},{y,y0},{z,z0},...? The secret is in the code below:

```
FindRoot[equations,Evaluate[
   Apply[Sequence,
     Transpose[{unknowns,initguesses}] ]] ];
```

To see how this works, take
unknowns = {x,y,z},initguesses = {1,1,1}. The above code generates

```
Evaluate[Sequence @@ Transpose[{{x, y, z}, {1, 5, 10}}]]
```

```
Sequence[{x, 1}, {y, 5}, {z, 10}]
```

This is the required input to FindRoot.

```
FindRoot[{x + y + z == 1, x == Sin[x] Cos[z], x == z},
 Evaluate[Sequence @@ Transpose[{{x, y, z}, {1, 5, 10}}]]]
```

```
FindRoot::lstol: The line search decreased the step size to within tolerance specified by AccuracyGoal and
PrecisionGoal but was unable to find a sufficient
     decrease in the merit function.  You may need more than MachinePrecision digits of working precision to
meet these tolerances. More…
```

```
{x -> 1.94659 10^-6, y -> 0.999996, z -> 1.94659 10^-6}
```

■ Attributes

There are several attributes that may be attached to *Mathematica* lists. (For details, see Introduction to Numerical Computing, Robert Skeel and Jerry Keiper.) For user-created functions, a useful attribute is Listable. Consider the following function f, which is to operate on a list:

```
Clear[f]
f[Range[10]]
```

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

What we wanted was to map f onto each element of the list. Checking the attributes of f:

```
Attributes[f]
```

```
{}
```

```
Attributes[f] = {Listable};
f[Range[10]]
```

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

An application of this occured in our Romberg Integration package, where we begin with a list of domain points (say, {a,b}) and want f of this, that is, f[{a,b}] = {f[a],f[b]}. Most *Mathematica* functions automatically have this property.

■ Caching

Caching is very valuable for speeding up evaluation when using recursive definitions. For example, the Fibonacci numbers are:

```
Clear[fib]
fib[n_] := fib[n - 1] + fib[n - 2];
fib[0] = fib[1] = 1;
Timing[fib[15]]
Timing[fib[16]]
```

```
{1.8166666666666668206 Second, 987}
```

```
{2.949999999999999289 Second, 1597}
```

Caching is a process where the intermediate values are stored in a table. The first call may sometimes take longer than non-caching for this reason, but subsequent calls will be take much less time.

```
Clear[fib]
fib[n_] := fib[n] = fib[n - 1] + fib[n - 2];
fib[0] = fib[1] = 1;
Timing[fib[15]]
Timing[fib[16]]
```

```
{0.06666666666666642982 Second, 987}
```

```
{0. Second, 1597}
```

Another application of a recursive scheme where caching yields impressive time savings is Romberg Integration, in which

```
R[i_, j_] := R[i, j] = N[(4^(j-1) R[i, j - 1] - R[i - 1, j - 1])/(4^(j-1) - 1)];
R[i_, 1] := R[i, 1] = TrapezoidRule[RangePoints, i];
```

■ More Efficient Programming Style should speed program AND Decrease Memory.

Using Share[] reduces memory use for symbols that are accessed repeatedly. Share[] has the effect of overwriting the old definitions rather than saving them. To see this, we'll use MemoryInUse[] as a tool to compare performance with and without Share.

```
startmemory = MemoryInUse[]
Timing[Sum[Erf[1.]^i, {i, 0, 10}]]
MemoryInUse[] - startmemory
```

```
810780
```

```
{0.08333333333333348137 Second, 5.389744472249423219}
```

```
1152
```

Share won't work on this since it is a question of inefficient programming style. Can use Horner's Rule:

```
Clear[HornersRule]
HornersRule[f_, EndingValue_] := Module[{}, Last[NestList[1 + f #1 &, 1 + f, EndingValue]]]

startmemory = MemoryInUse[]
Timing[HornersRule[Erf[1.], 9]]
MemoryInUse[] - startmemory
```

```
872908
```

```
{0.01666666666666660745 Second, 5.389744472249423218}
```

```
1088
```

■ MemoryInUse[]

This tool is useful for understanding why your program may be running slowly or your system eventually crashes, especially when rendering large, complex graphics. What it does:

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

```
MemoryInUse[ ] gives the number of bytes
   currently being used to store all data in the
   current Mathematica session.
```

See WRI Technical Note by Shawn Sheridan (*MathSource*) about Memory for more information.

He gives the example of creating a large list and trying to clear it:

```
startmemory = MemoryInUse[];
bigList = Range[10^5];
MemoryInUse[] - startmemory
```

```
1598924
```

Clearing the list does not release the memory allocated to it until the number of pointers to the symbol go to zero:

```
Clear[bigList]
MemoryInUse[] - startmemory
```

```
1601076
```

```
Unprotect[In, Out]
Clear[In, Out];
MemoryInUse[] - startmemory
```

```
{}
```

```
3860
```

■ A List of Constant Length

```
Clear[UpdatedTemps]
startmemory = MemoryInUse[];
Temperatures = Table[30, {10}];
Do[UpdatedTemps = Take[Temperatures, Length[Temperatures] - 1] +
   Table[Random[], {Length[Temperatures] - 1}];
 Temperatures = AppendTo[UpdatedTemps, UpdatedTemps[[Length[Temperatures] - 2]]];
 Print["Length of list = ", Length[UpdatedTemps]];
 Print[MemoryInUse[] - startmemory], {10}]
```

■ Using AppendTo May Increase Memory Usage

```
Clear[UpdatedTemps]
startmemory = MemoryInUse[];
Temperatures = Table[30, {10}];
UpdatedTemps := Take[Temperatures, Length[Temperatures] - 1] +
   Table[Random[], {Length[Temperatures] - 1}];
Do[Temperatures = AppendTo[UpdatedTemps, UpdatedTemps[[Length[Temperatures] - 2]]];
 Print["Length of list = ", Length[UpdatedTemps]];
 Print[MemoryInUse[] - startmemory], {10}]
```

If you have enough variables like this, your program may eventually reach the machine limit and crash. In this case, the memory is increasing linearly.

■ EnterDialog[]

■ Compiling

Once your *Mathematica* code is written, you can speed it up in several steps, one of which is CompiledFunction. However, this will only speed up your code when the number of external symbol calls is minimal, ideally zero. The following example should illustrate this:

```
Clear[f]
f[x_, y_] := 2 x / (x^2 + y^2)
Timing[Table[f[x, y], {x, 100}, {y, 100}];]
Clear[f]
f = Compile[{x, y}, 2 x / (x^2 + y^2)];
Timing[Table[f[x, y], {x, 100}, {y, 100}];]
```

```
{33.08333333333333348 Second, Null}
```

```
{9.51666666666666608 Second, Null}
```

NOTE: You cannot use Compile directly with lists. Additionally, be aware of the overhead costs involved with Compile. Compile doesn't always speed operations up:

```
Timing[NIntegrate[x, {x, 0, 10}, Compiled → False]]
Timing[NIntegrate[x, {x, 0, 10}, Compiled → True]]
```

```
{0.016666666666668295 Second, 50.00000000000000001}
```

```
{0.033333333333343695 Second, 50.00000000000000001}
```

■ FullForm

■ IBM File Format

Tricks

■ String Manipulation (ToExpression,ToString,<<)

■ Keeping the Leading Zeros

I wanted to keep track of my experimental data files, which begin with the string "L108" and are followed by a four digit number, starting with 0000. For demonstration, I'll set up and use fileNames[], but in practice this would be the actual FileNames[].

```
fileNames[] =
  {"Animation of Mod BA Surface", "anything.ma", "BAOUT12.GRD", "BAOUT24.GRD",
   "BAOUT50.GR     D", "BAOUT6.GRD", "BAOUT75.GRD", "BA     Version 2 50 mm Results",
   "BAV2_12.ma", "bav224cp.ma", "bav250.ma", "bav26cp.ma", "bav275.ma",
   "BA_06.DAT", "BA_12.DAT", "BA_24.DAT", "BA_50.DAT", "BA_75.DAT", "HTFOIL2.CFG",
   "L1080000", "L1080001", "L1080002", "L1080003", "L1080004", "L108     0005",
   "L1080006", "Mod BA v2 75 mm     results", "Module BA V2 12 mm Results",
   "Module BA V2 6 iwg", "Module BA v2 6 mm     Results", "Module BA 24 mm Results",
   "plots     .ma", "0001", "0002", "0003", "0004", "000     5"};
```

What I want to do is identify all files starting with the string "L108" and eight characters long. This test is not foolproof yet and needs more work, but will give the idea. Here's what I want the application to do:

```
listofeligiblefiles =
  Select[fileNames[], StringTake[#1, 4] == "L108" && StringLength[#1] == 8 &];
LatestL108File$ = StringTake[Last[Sort[listofeligiblefiles]], {5, 8}];
inputpromptstring = "The last test number used was " <>
   LatestL108File$ <> "\nSave as next number (y/n): ";
FileSaveResp$ = Input[inputpromptstring];

padnumber[x_] := If[Length[IntegerDigits[x]] < 4,
  Table["0", {4 - Length[IntegerDigits[x]]}] <> ToString[x], x]
If[ToLowerCase[ToString[FileSaveResp$]] === "y",
  FileSaveName$ = currentdir <> "L108" <> StringTake[
     ToString[ToExpression["." <> StringTake[LatestL108File$, 4]] + .00011], {3, 6}],
  FileSaveName$ = Input["Enter four digit number for file: "];
  FileSaveName$ = currentdir <> ToString[padnumber[FileSaveName$]]];
```

```
FileSaveName$
```

```
Quadra 800 HD:glasstech:L108:L108    Images:L1080007
```

NOTE: I had to create a file L1080000 to get things started.
This works by first finding a list of files which match the pattern L108xxxx. This list is stored as **listofeligible-file**s.

```
listofeligiblefiles
```

```
{L1080000, L1080001, L1080002, L108000     3, L1080004, L1080006}
```

Next, I want to strip off the last entry on this list and add 1 to the numeric xxxx field. I can sort, take the Last and use StringTake to identify this number.

```
StringTake[Last[Sort[listofeligiblefiles]], {5, 8}]
```

```
0006
```

However, I can't add 1 to a string. If I convert this to a number so addition is possible, I get:

```
ToExpression[%]
```

```
6
```

My solution was to do the addition anyway, but recover the original number by converting back to a string, comparing the number of digits in the numeric value with the original number of digits and pad with zeros.

```
StringTake[
 ToString[ToExpression["." <> StringTake[LatestL108File$, 4]] +
    .00011], {3, 6}]
```

```
0007
```

How does this work?

```
StringTake[LatestL108File$, 4]
```

```
0006
```

To save the leading zeros, put a decimal point at the beginning of this number:

```
ToExpression["." <> StringTake[LatestL108File$, 4]]
```

```
0.0006
```

Now we can add:

```
% + .00011
```

```
0.00071
```

Convert back to a string, and take elements 3 through 6:

```
StringTake[ToString[ToExpression["." <> StringTake[LatestL108File$, 4]] + .00011], {3, 6}]
```

```
0007
```

Something else that I needed was to allow the user to select a number, say 11, and turn this into 0011. I did this by creating a function "padnumber[x]" which counts the number of digits in the user input, converts to a string and writes however many 0's are needed to fill out the entry to a four digit number.

```
padnumber[x_] := If[Length[IntegerDigits[x]] < 4,
  Table["0", {4 - Length[IntegerDigits[x]]}] <> ToString[x], x]
```

```
padnumber[1]
```

```
0001
```

```
padnumber[1000]
```

```
1000
```

```
padnumber[6]
```

```
0006
```

```
IntegerDigits[6]
```

```
{6}
```

■ 0 vs 0.

This problem has come up several times in various applications that I have written. The problem is that 0 is not the same as 0., which can be seen by examining the Head:

```
{Head[0], Head[0.]}
```

```
{Integer, Real}
```

To understand the problem, note what happens when we multiply:

```
{0 kJoule, 0. kJoule}
```

```
{0, 0. kJoule}
```

Can we change the Head of a primitive?

```
Real[0]
```

```
Real[0]
```

Not likely. There are at least two things to try: Either redefine the properties of 0, or always work with 0.

Another problem is that 0000 and the like are truncated:

```
{0, 19}
```

```
{0, 19}
```

The relevant applications are: (1) Reading in data and units (for example, Thermodynamics property data); (2) Experimental Report Writer which keeps track of latest report number (say, 0000) and automatically increments by 1 if user desires.

■ Thermodynamics Property Data

The following sample code demonstrates how to attach unit values to imported data, in this case thermodynamic steam table data, for your own applications. The moral is to never use the integer 0, use 0. instead.

```
Needs["Engineering`Thermodynamics`"]
```

```
interpolate[PressureTable, P == 0.2]
```

$$\{\text{P} \to 0.2\ \text{Mega Pascal},\ \text{T} \to 120.23\ \text{Celsius},\ \text{vf} \to \frac{0.001061\ \text{m}^3}{\text{kg}},\ \text{vfg} \to \frac{0.884639\ \text{m}^3}{\text{kg}},$$
$$\text{vg} \to \frac{0.8857\ \text{m}^3}{\text{kg}},\ \text{uf} \to \frac{504.49\ \text{kJ}}{\text{kg}},\ \text{ufg} \to \frac{2025.\ \text{kJ}}{\text{kg}},\ \text{ug} \to \frac{2529.5\ \text{kJ}}{\text{kg}},\ \text{hf} \to \frac{504.7\ \text{kJ}}{\text{kg}},$$
$$\text{hfg} \to \frac{2201.9\ \text{kJ}}{\text{kg}},\ \text{hg} \to \frac{2706.7\ \text{kJ}}{\text{kg}},\ \text{sf} \to \frac{1.5301\ \text{kJ}}{\text{K kg}},\ \text{sfg} \to \frac{5.597\ \text{kJ}}{\text{K kg}},\ \text{sg} \to \frac{7.1271\ \text{kJ}}{\text{K kg}}\}$$

This was done by the following sequence of commands:
The above numbers represent the pressure, temperature, vf, vg, uf, ufg, ug, hf, hfg, hg, sf, sfg, sg of the water. We want to associate each of the above values with appropriate units and have rules such as {P -> 0.0006113 MPa, T -> 0.01 C, ... }.
Let us begin by defining lists of property names and units in order:

```
listofproperties = {P, T, vf, vg, uf, ufg, ug, hf, hfg, hg, sf, sfg, sg};
units =
  {Mega Pascal, Celsius, m^3/kg, m^3/kg, kJ/kg, kJ/kg, kJ/kg, kJ/kg, kJ/kg, kJ/kg, kJ/(kg K), kJ/(kg K), kJ/(kg K)};
interpolate[x_List] := Module[{}, (#1[[1]] → #1[[2]] &) /@ Transpose[
    {listofproperties, MapThread[Times, {(#1[[1]] &) /@ Transpose[{x, Units}], units}]}]]
```

First, we will match up the units with the numeric property values.
Taking the transpose yields:

```
Transpose[{{0.2, 120.23, .001061, .8857, 504.49,
   2025., 2529.5, 504.7, 2201.9, 2706.7, 1.5301, 5.597, 7.1271}, units}]
```

$$\left\{\{0.2,\ \text{Mega Pascal}\},\ \{120.23,\ \text{Celsius}\},\ \left\{0.001061,\ \frac{\text{m}^3}{\text{kg}}\right\},\ \left\{0.8857,\ \frac{\text{m}^3}{\text{kg}}\right\},\ \left\{504.49,\ \frac{\text{kJ}}{\text{kg}}\right\},\ \left\{2025.,\ \frac{\text{kJ}}{\text{kg}}\right\},\ \left\{2529.5,\ \frac{\text{kJ}}{\text{kg}}\right\},\ \left\{504.7,\ \frac{\text{kJ}}{\text{kg}}\right\},\ \left\{2201.9,\ \frac{\text{kJ}}{\text{kg}}\right\},\ \left\{2706.7,\ \frac{\text{kJ}}{\text{kg}}\right\},\ \left\{1.5301,\ \frac{\text{kJ}}{\text{K kg}}\right\},\ \left\{5.597,\ \frac{\text{kJ}}{\text{K kg}}\right\},\ \left\{7.1271,\ \frac{\text{kJ}}{\text{K kg}}\right\}\right\}$$

This is not quite what we need: we actually want something like a multiplication of the property value and unit. Try the following:

```
(#1[[1]] → #1[[2]] &) /@ Transpose[{listofproperties,
   MapThread[Times, {(#1[[1]] &) /@ Transpose[{PressureTable[[1]], Units}], Units}]}]
```

$$\left\{\text{P} \to 0.0006113\ \text{MPa},\ \text{T} \to 0.01\ \text{Celsius},\ \text{vf} \to \frac{0.001\ \text{m}^3}{\text{kg}},\ \text{vg} \to \frac{206.14\ \text{m}^3}{\text{kg}},\ \text{uf} \to 0,\ \text{ufg} \to \frac{2375.3\ \text{kJ}}{\text{kg}},\ \text{ug} \to \frac{2375.3\ \text{kJ}}{\text{kg}},\ \text{hf} \to \frac{0.01\ \text{kJ}}{\text{kg}},\ \text{hfg} \to \frac{2501.3\ \text{kJ}}{\text{kg}},\ \text{hg} \to \frac{2501.4\ \text{kJ}}{\text{kg}},\ \text{sf} \to 0,\ \text{sfg} \to \frac{9.156200000000000001\ \text{kJ}}{\text{K kg}},\ \text{sg} \to \frac{9.156200000000000001\ \text{kJ}}{\text{K kg}}\right\}$$

```
interpolate[x_List] := Module[{}, (#1[[1]] → #1[[2]] &) /@ Transpose[
    {listofproperties, MapThread[Times, {(#1[[1]] &) /@ Transpose[{x, units}], units}]}]]

interpolate[TemperatureTable, T == 100]
```

$$\left\{\text{T} \to 100\ \text{Celsius},\ \text{P} \to 0.10135\ \text{Mega Pascal},\ \text{vf} \to \frac{0.001044\ \text{m}^3}{\text{kg}},\ \text{vg} \to \frac{1.6729\ \text{m}^3}{\text{kg}},\ \text{uf} \to \frac{418.94\ \text{kJ}}{\text{kg}},\ \text{ufg} \to \frac{2087.6\ \text{kJ}}{\text{kg}},\ \text{ug} \to \frac{2506.5\ \text{kJ}}{\text{kg}},\ \text{hf} \to \frac{419.04\ \text{kJ}}{\text{kg}},\ \text{hfg} \to \frac{2257.\ \text{kJ}}{\text{kg}},\ \text{hg} \to \frac{2676.1\ \text{kJ}}{\text{kg}},\ \text{sf} \to \frac{1.3069\ \text{kJ}}{\text{K kg}},\ \text{sfg} \to \frac{6.048\ \text{kJ}}{\text{K kg}},\ \text{sg} \to \frac{7.3549\ \text{kJ}}{\text{K kg}}\right\}$$

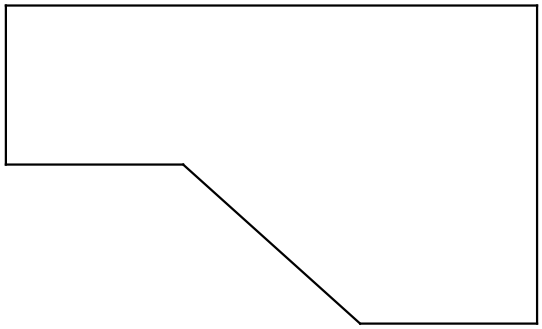
■ Matrix Manipulations

Not finished yet ... writing as a package

■ **Potential Flow**

Problem Description

The flow conduit shown below has dimensions of 1.6 m x 1.2 m.
The conduit entrance is 0.5 meters wide. The conduit expands along a slanted surface which starts 0.6 m along the plate and 0.6 m high down to the point 1.2 m along the conduit bottom. The exit is 1.2 m wide.
A perfect fluid enters the conduit at a velocity of 10 m/s along the free surface.
The flow velocity is zero along the lower surface of the conduit, and varies from zero to 10 m/s along the entrance and exit planes.
The velocity distribution of the fluid inside the conduit is desired.



Analysis

Potential flow conditions are assumed. In this case, the Laplace equation

Although we are solving for velocities here, the same equation governs temperature and electrostatic distributions.

The most difficult aspect of this problem is the description of the boundary surface. At least two possibities suggest themselves: (i) use Mathematica's Table function to build up a list of points; (ii) define functions that describe the boundary. Here, we show the first method since it seems more natural in Mathematica.
The solution will proceed in the following steps:

1. Discretize the domain
2. Write the finite-difference form of the Laplace equation (FDE) and boundary conditions.
3. The FDE is written at each node at which the solution is unknown (all interior points).
4. The resulting system of linear equations is solved and plotted.

■ Parameter and Boundary Definitions

```
Off[General::"spell1"]
Off[General::"spell"]
Clear[u]
LengthofPlate = 1.6; Deltax = 0.2;
n = Ceiling[LengthofPlate / Deltax] + 1;
HeightofPlate = 1.2; Deltay = 0.2;
m = Ceiling[HeightofPlate / Deltay] + 1;
boundary = {{1, m}, {n, m}, {n, 1}, {m, 1}, {4, 4}, {1, 4}, {1, m}};
geometry = Line /@ Partition[boundary, 2, 1];
boundarySurface = Graphics[geometry];
```

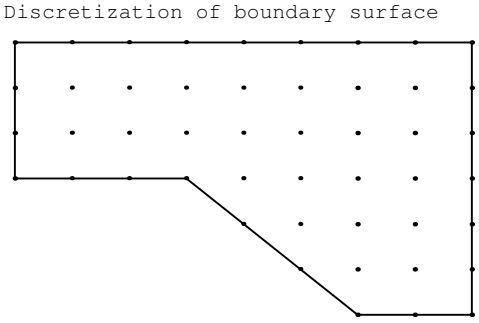
Discretization

Describing the surface takes some experimentation. For our particular geometry, the following code will generate the nodal points:

```
nodalpoints = Join[Table[Table[{i, j}, {i, n}], {j, m, 4, -1}],
   Table[Table[{i, j}, {i, 8 - j, n}], {j, 3, 1, -1}]];
```

In order to plot our points, we have to wrap **Point[]** around each pair. One way to accomplish this is via the following transformation rule. We then wrap **Graphics[]** around the result and name the whole thing "discretization" for later reference:

```
discretization = Graphics[nodalpoints /. {x_, y_} :> Point[{x, y}]];
Show[boundarySurface, discretization,
  PlotLabel → " Discretization of boundary surface"];
```



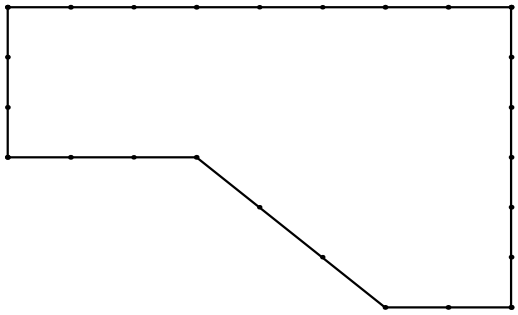
We now wish to write the Laplace equation at each point where the flow field is unknown. The field is unknown at all of the interior points shown above. Since we already have these defined as nodalpoints, we should use them. The flow velocity is given along the boundaries, so the unknown values are nodalpoints minus our boundaries.

```
Boundarys := Join[Table[u[i, m], {i, n}], Table[u[n, j], {j, m}], Table[u[i, 4], {i, 4}],
   Table[u[i, 1], {i, m, n}], Table[u[i + 4, -i + 4], {i, 2}], Table[u[1, j], {j, 4, m}]];
InteriorPoints = Complement[Flatten[nodalpoints /. {x_, y_} :→ u[x, y]], Boundarys];
```

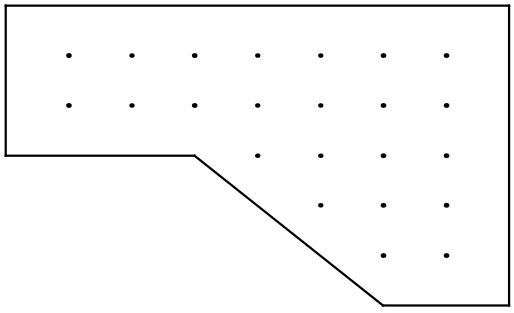
A graphical display is the easiest way to check our work:

Domain Plots

```
Show[Graphics[Boundarys /. u[x_, y_] :→ Point[{x, y}]], boundarySurface];
```



```
Show[Graphics[InteriorPoints /. u[x_, y_] :→ Point[{x, y}]], boundarySurface];
```



Finite Difference Equation

The finite difference approximation of the Laplace Equation becomes

```
equation[i_, j_] = Simplify[
  (u[i + 1, j] - 2 u[i, j] + u[i - 1, j])/Deltax^2 + 1/Deltay^2 (u[i, j + 1] - 2 u[i, j] + u[i, j - 1]) == 0]
```

```
25. u[-1 + i, j] + 25. u[i, -1 + j] - 100. u[i, j] + 25. u[i, 1 + j] + 25. u[1 + i, j] == 0
```

The given boundary conditions are:

```
Boundarys := Join[Table[u[i, m], {i, n}], Table[u[n, j], {j, m}], Table[u[i, 4], {i, 4}],
   Table[u[i, 1], {i, m, n}], Table[u[i + 4, -i + 4], {i, 2}], Table[u[1, j], {j, 4, m}]];

u[i_, m] := 10 /; i ≥ 1 && i ≤ n;
u[n, j_] := 10 (j - 1)/6 /; j ≥ 1 && j ≤ m;
u[i_, 4] := 0 /; i ≥ 1 && i ≤ 4;
u[i_, 1] := 0 /; i ≥ m && i ≤ n;
u[1, j_] := 10 (j - 4)/3; j ≥ 4 && j ≤ m;
Do[u[i + 4, 4 - i] := 0, {i, 2}];

nodalpoints /. {x_, y_} :→ u[x, y]
```

$$\Big\{\{10, 10, 10, 10, 10, 10, 10, 10, 10\},\ \Big\{\frac{20}{3}, u[2, 6], u[3, 6], u[4, 6], u[5, 6], u[6, 6], u[7, 6], u[8, 6], \frac{25}{3}\Big\},\ \Big\{\frac{10}{3}, u[2, 5], u[3, 5], u[4, 5], u[5, 5], u[6, 5], u[7, 5], u[8, 5], \frac{20}{3}\Big\},\ \{0, 0, 0, 0, u[5, 4], u[6, 4], u[7, 4], u[8, 4], 5\},\ \Big\{0, u[6, 3], u[7, 3], u[8, 3], \frac{10}{3}\Big\}, \Big\{0, u[7, 2], u[8, 2], \frac{5}{3}\Big\}, \{0, 0, 0\}\Big\}$$

Write the list of equations:

```
InteriorPoints /. u[i_, j_] :→ equation[i, j]
```

```
{83.33333333333333333 - 100. u[2, 5] + 25. u[2, 6] + 25. u[3, 5] == 0,
 416.66666666666666 + 25. u[2, 5] - 100. u[2, 6] + 25. u[3, 6] == 0,
 25. u[2, 5] - 100. u[3, 5] + 25. u[3, 6] + 25. u[4, 5] == 0,
 250. + 25. u[2, 6] + 25. u[3, 5] - 100. u[3, 6] + 25. u[4, 6] == 0,
 25. u[3, 5] - 100. u[4, 5] + 25. u[4, 6] + 25. u[5, 5] == 0,
 250. + 25. u[3, 6] + 25. u[4, 5] - 100. u[4, 6] + 25. u[5, 6] == 0,
 -100. u[5, 4] + 25. u[5, 5] + 25. u[6, 4] == 0,
 25. u[4, 5] + 25. u[5, 4] - 100. u[5, 5] + 25. u[5, 6] + 25. u[6, 5] == 0,
 250. + 25. u[4, 6] + 25. u[5, 5] - 100. u[5, 6] + 25. u[6, 6] == 0,
 -100. u[6, 3] + 25. u[6, 4] + 25. u[7, 3] == 0,
 25. u[5, 4] + 25. u[6, 3] - 100. u[6, 4] + 25. u[6, 5] + 25. u[7, 4] == 0,
 25. u[5, 5] + 25. u[6, 4] - 100. u[6, 5] + 25. u[6, 6] + 25. u[7, 5] == 0,
 250. + 25. u[5, 6] + 25. u[6, 5] - 100. u[6, 6] + 25. u[7, 6] == 0,
 -100. u[7, 2] + 25. u[7, 3] + 25. u[8, 2] == 0,
 25. u[6, 3] + 25. u[7, 2] - 100. u[7, 3] + 25. u[7, 4] + 25. u[8, 3] == 0,
 25. u[6, 4] + 25. u[7, 3] - 100. u[7, 4] + 25. u[7, 5] + 25. u[8, 4] == 0,
 25. u[6, 5] + 25. u[7, 4] - 100. u[7, 5] + 25. u[7, 6] + 25. u[8, 5] == 0,
 250. + 25. u[6, 6] + 25. u[7, 5] - 100. u[7, 6] + 25. u[8, 6] == 0,
 41.66666666666666666 + 25. u[7, 2] - 100. u[8, 2] + 25. u[8, 3] == 0,
 83.33333333333333333 + 25. u[7, 3] + 25. u[8, 2] - 100. u[8, 3] + 25. u[8, 4] == 0,
 125. + 25. u[7, 4] + 25. u[8, 3] - 100. u[8, 4] + 25. u[8, 5] == 0,
 166.66666666666666667 + 25. u[7, 5] + 25. u[8, 4] - 100. u[8, 5] + 25. u[8, 6] == 0,
 458.33333333333333333 + 25. u[7, 6] + 25. u[8, 5] - 100. u[8, 6] == 0}
```

Ideally, we would at this point just call **Solve** with the above and the list of unknowns **InteriorPoints** as inputs. However, the built-in **Solve** function will take too long on this system, although it works well on smaller systems. We'd like to put the system into matrix form. The ***Mathematica*** function **CoefficientList** would seem to be ideal for this

task; unfortunately, it does not work this way! I'd like to compare each expression with **InteriorPoints** and pick off the matching coefficients, entry by entry. This problem occurs over and over in solving these systems of equations.

The first step in our solution is to turn the output from **Solve** into a list. This is easily done by typing:

```
equationlist =
 Table[Flatten[%[[i]] /. {Equal → List, Plus → List}], {i, Length[InteriorPoints]}]
```

```
{{83.33333333333333333, -100. u[2, 5], 25. u[2, 6], 25. u[3, 5], 0},
 {416.6666666666666666, 25. u[2, 5], -100. u[2, 6], 25. u[3, 6], 0},
 {25. u[2, 5], -100. u[3, 5], 25. u[3, 6], 25. u[4, 5], 0},
 {250., 25. u[2, 6], 25. u[3, 5], -100. u[3, 6], 25. u[4, 6], 0},
 {25. u[3, 5], -100. u[4, 5], 25. u[4, 6], 25. u[5, 5], 0},
 {250., 25. u[3, 6], 25. u[4, 5], -100. u[4, 6], 25. u[5, 6], 0},
 {-100. u[5, 4], 25. u[5, 5], 25. u[6, 4], 0},
 {25. u[4, 5], 25. u[5, 4], -100. u[5, 5], 25. u[5, 6], 25. u[6, 5], 0},
 {250., 25. u[4, 6], 25. u[5, 5], -100. u[5, 6], 25. u[6, 6], 0},
 {-100. u[6, 3], 25. u[6, 4], 25. u[7, 3], 0},
 {25. u[5, 4], 25. u[6, 3], -100. u[6, 4], 25. u[6, 5], 25. u[7, 4], 0},
 {25. u[5, 5], 25. u[6, 4], -100. u[6, 5], 25. u[6, 6], 25. u[7, 5], 0},
 {250., 25. u[5, 6], 25. u[6, 5], -100. u[6, 6], 25. u[7, 6], 0},
 {-100. u[7, 2], 25. u[7, 3], 25. u[8, 2], 0},
 {25. u[6, 3], 25. u[7, 2], -100. u[7, 3], 25. u[7, 4], 25. u[8, 3], 0},
 {25. u[6, 4], 25. u[7, 3], -100. u[7, 4], 25. u[7, 5], 25. u[8, 4], 0},
 {25. u[6, 5], 25. u[7, 4], -100. u[7, 5], 25. u[7, 6], 25. u[8, 5], 0},
 {250., 25. u[6, 6], 25. u[7, 5], -100. u[7, 6], 25. u[8, 6], 0},
 {41.66666666666666666, 25. u[7, 2], -100. u[8, 2], 25. u[8, 3], 0},
 {83.33333333333333333, 25. u[7, 3], 25. u[8, 2], -100. u[8, 3], 25. u[8, 4], 0},
 {125., 25. u[7, 4], 25. u[8, 3], -100. u[8, 4], 25. u[8, 5], 0},
 {166.6666666666666667, 25. u[7, 5], 25. u[8, 4], -100. u[8, 5], 25. u[8, 6], 0},
 {458.3333333333333333, 25. u[7, 6], 25. u[8, 5], -100. u[8, 6], 0}}
```

The following code will match up the unknowns with their coefficients from **Solve**. All we have to do is worry about the right hand sides. This code is general and will work with any list of equations (**equationlist**) and coefficient lists (**InteriorPoints**).

WARNING! Order in the Complement function matters! Look at the two different answers I get by just changing the order of targuments

First, a sample of **Coefficient**:

```
Coefficient[{1, x, x^2, x^3}, x]
```

```
{0, 1, 0, 0}
```

Now take the complement with a list of zeros. The complement function gives the list of elements of the first argument which are not in any list. First, look for the list of items in
{1,2,3} that are NOT in {0,1,0,0}:

```
Complement[{1, 2, 3}, Coefficient[{1, x, x^2, x^3}, x]]
```

```
{2, 3}
```

This is the answer we expected. Reversing the order means something different to Complement: we are looking for the items in {0,1,0,0} which are NOT in {1,2,3}.

```
Complement[Coefficient[{1, x, x^2, x^3}, x], {1, 2, 3}]
```

```
{0}
```

Generation of Matrix Coefficients from Equations

```
list = {};
zerolist = Table[0, {Length[InteriorPoints]}];
Do[Do[If[Complement[Coefficient[equationlist[[j]], InteriorPoints[[i]]], zerolist] === {},
   AppendTo[list, 0], AppendTo[list,
    Complement[Coefficient[equationlist[[j]], InteriorPoints[[i]]], zerolist]]],
  {i, Length[InteriorPoints]}], {j, Length[InteriorPoints]}]
list = Partition[Flatten[list], Length[InteriorPoints]]
```

```
{{-100., 25., 25., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {25., -100., 0, 25., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {25., 0, -100., 25., 25., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 25., 25., -100., 0, 25., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 25., 0, -100., 25., 0, 25., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 25., 25., -100., 0, 0, 25., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, -100., 25., 0, 0, 25., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 25., 0, 25., -100., 25., 0, 0, 25., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 25., 0, 25., -100., 0, 0, 0, 25., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, -100., 25., 0, 0, 0, 25., 0, 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 25., 0, 0, 25., -100., 25., 0, 0, 0, 25., 0, 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 25., 0, 0, 25., -100., 25., 0, 0, 0, 25., 0, 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 25., 0, 0, 25., -100., 0, 0, 0, 0, 25., 0, 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -100., 25., 0, 0, 0, 25., 0, 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 25., 0, 0, 0, 25., -100., 25., 0, 0, 0, 25., 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 25., 0, 0, 0, 25., -100., 25., 0, 0, 0, 25., 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 25., 0, 0, 0, 25., -100., 25., 0, 0, 0, 25.},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 25., 0, 0, 0, 0, -100., 25., 0, 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 25., 0, 0, 0, 25., -100., 25., 0, 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 25., 0, 0, 0, 25., -100., 25., 0},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 25., 0, 0, 0, 25., -100., 25.},
 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 25., 0, 0, 0, 25., -100.}}
```

To get the right hand side, note that if a constant appears, it is always the first term.
Look at all of the first elements of each equation. There are two cases which can occur: either a number appears, in which case, we want it in our list of right hand side coefficients (of course, moving it to the right hand side changes its sign); or a coefficient of an unknown u[i,j] appears. If this is the case, we want the corresponding right hand side value to be zero

(since that is the only constant appearing. Looking at the **FullFor**m of a typical equation gave us the following idea to do as described:

```
rhs = Table[-equationlist[[i]][[1]], {i, Length[InteriorPoints]}] /. x_ y_ :> 0
```

```
{-83.33333333333333333, -416.66666666666666, 0, -250., 0, -250.,
 0, 0, -250., 0, 0, 0, -250., 0, 0, 0, 0, -250., -41.66666666666666666,
 -83.33333333333333333, -125., -166.66666666666666667, -458.33333333333333333}
```

Since our matrix is not tridiagonal, we can't take advantage of the speed of the **LinearSolve** function in the package LinearAlgebra. We will solve by matrix inversion instead.

```
inversematrix = Inverse[list];

solution = inversematrix.rhs;
```

We now want to map our solution back onto the nodal points for plotting:

```
nodalpoints = nodalpoints /. {x_, y_} :> u[x, y]
```

$$\left\{\{10, 10, 10, 10, 10, 10, 10, 10, 10\},\ \left\{\frac{20}{3}, u[2, 6], u[3, 6], u[4, 6], u[5, 6], u[6, 6], u[7, 6], u[8, 6], \frac{25}{3}\right\},\ \left\{\frac{10}{3}, u[2, 5], u[3, 5], u[4, 5], u[5, 5], u[6, 5], u[7, 5], u[8, 5], \frac{20}{3}\right\},\ \{0, 0, 0, 0, u[5, 4], u[6, 4], u[7, 4], u[8, 4], 5\},\ \left\{0, u[6, 3], u[7, 3], u[8, 3], \frac{10}{3}\right\}, \left\{0, u[7, 2], u[8, 2], \frac{5}{3}\right\}, \{0, 0, 0\}\right\}$$

Substitute the solution in for the unknowns.

```
solution = nodalpoints /. MapThread[Rule, {InteriorPoints, solution}]
```

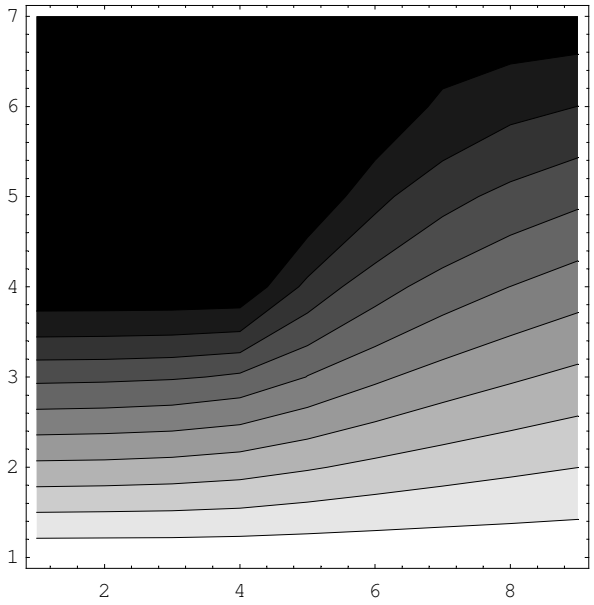
$$\left\{\{10, 10, 10, 10, 10, 10, 10, 10, 10\}, \left\{\frac{20}{3}, 6.707960708039365723, 6.785234648417354938, 6.954505858708987932, 7.284076844225128642, 7.619931635047708061, 7.903328030670796375, 8.133501784950637739, \frac{25}{3}\right\},\ \left\{\frac{10}{3}, 3.379941517073441288, 3.478472026921066094, 3.748711942193468151, 4.561869883143818578, 5.292321665294907228, 5.8598787026848397, 6.297345775798421243, \frac{20}{3}\right\}, \{0, 0, 0, 0, 1.922369080861770287, 3.127606440303262571, 3.946519338975233956, 4.529335948891540865, 5\},\ \left\{0, 1.349215676081138814, 2.269256264021292686, 2.873478680792508261, \frac{10}{3}\right\},\ \left\{0, 0.9078113602362897114, 1.36198917692386616, \frac{5}{3}\right\}, \{0, 0, 0\}\right\}$$

We now have a slight problem. ***Mathematica*'s ListPlot3D** function will only work with rectangular arrays of data. To get around this, we can insert 0's where the boundary wall appears (the flow would be zero there anyway of course!) This method could be adapted to derivative boundary conditions, but seems as though it would not work for convection-type boundary conditions (boundary conditions of the Third Type).

```
VelocityPoints = Partition[
  Flatten[{Table[solution[[i]], {i, 4}], Flatten[PrependTo[solution[[5]], {0, 0, 0, 0}]],
    Flatten[PrependTo[solution[[6]], {0, 0, 0, 0, 0}]],
    Flatten[PrependTo[solution[[7]], {0, 0, 0, 0, 0, 0}]]}], 9]
```

```
{{10, 10, 10, 10, 10, 10, 10, 10, 10}, {20/3, 6.707960708039365723,
  6.785234648417354938, 6.954505858708987932, 7.284076844225128642,
  7.619931635047708061, 7.903328030670796375, 8.133501784950637739, 25/3},
 {10/3, 3.379941517073441288, 3.478472026921066094, 3.748711942193468151,
  4.561869883143818578, 5.292321665294907228, 5.8598787026848397,
  6.297345775798421243, 20/3}, {0, 0, 0, 0, 1.922369080861770287,
  3.127606440303262571, 3.946519338975233956, 4.529335948891540865, 5},
 {0, 0, 0, 0, 0, 1.349215676081138814, 2.269256264021292686, 2.873478680792508261, 10/3},
 {0, 0, 0, 0, 0, 0, 0.9078113602362897114, 1.36198917692386616, 5/3},
 {0, 0, 0, 0, 0, 0, 0, 0, 0}}
```

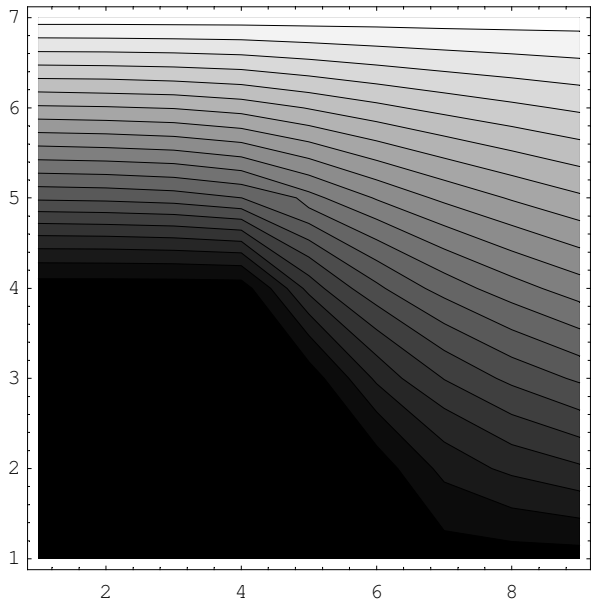
```
ListContourPlot[VelocityPoints];
```



For some reason, the picture is rendered upside down! Since there don't appear to be any ***Mathematica*** commands to invert a contour graphic, we inverted our data by using **Table** to put the data in the reverse order.

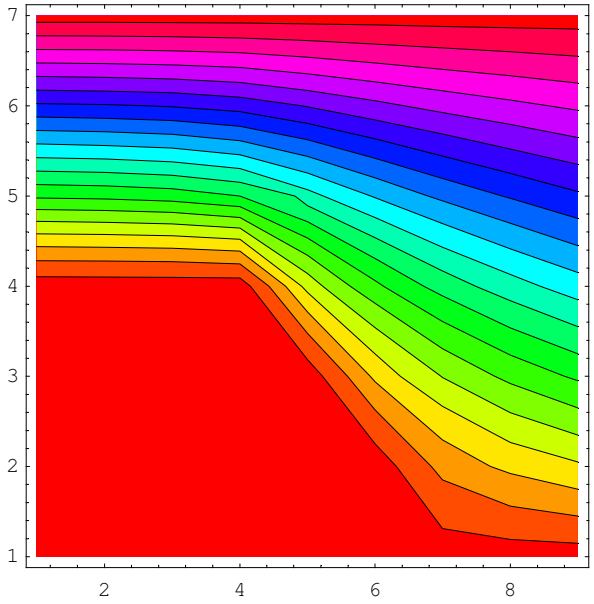
```
VelocityPoints = Table[VelocityPoints[[i]], {i, 7, 1, -1}];
```

```
ListContourPlot[VelocityPoints, Contours → 20];
```



It is kind of interesting to see the effects of our discretization in the slanted wall boundary above.

```
ListContourPlot[VelocityPoints, Contours → 20, ColorFunction → Hue];
```



■ Transferring Files Across Platforms

Keep filenames, etc. as generic as possible when writing for more than one platform. Use an .ini file to give current platform info. For example, on the IBM, a *Mathematica* program might have the command Open["c:\heat\results\filename"] but this has to be changed on the Mac. Or, might have SetDirectory["Macintosh HD:glasstech:Auxiliary Data Files:"] which has to be changed as I go from Mac to Mac. Instead, open a file that gives current file info each time. (Going from IBM to Mac: one thing I want to do is name a FOLDER c:\heat\results" ... but since the Mac uses : as a delimiter, doesn't work!!!)

Traps

■ Pattern Matching

We often have multi-dimensional lists and wish to identify specific portions. For example, the pairs {i,i^3} might be built up as:

```
xdata = Range[10];
ydata = xdata^3;
datalist = Transpose[{xdata, ydata}]
```

```
{{1, 1}, {2, 8}, {3, 27}, {4, 64}, {5, 125},
 {6, 216}, {7, 343}, {8, 512}, {9, 729}, {10, 1000}}
```

To identify the first or second points, we can use pattern matching as:

```
datalist /. {x_, y_} → x
```

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

However, what if x (or y) were previously defined?

```
x = 9;
datalist /. {x_, y_} → x
```

```
{9, 9, 9, 9, 9, 9, 9, 9, 9, 9}
```

A better way to do this might be to use a pure function, which needs no name.

```
(#1[[1]] &) /@ datalist
(#1[[2]] &) /@ datalist
```

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

```
{1, 8, 27, 64, 125, 216, 343, 512, 729, 1000}
```

■ Caching

Don't cache unless you are using recursion: if you are just going to be using your result once or twice, the extra time required may not pay off.

```
Clear[j]
j[x_] := j[x] = Erf[Sqrt[1 - x^2]] BesselJ[0, x]
Timing[Table[j[i], {i, 10}]]
```

```
{2.099999999999999645 Second,
 {0, i BesselJ[0, 2] Erfi[Sqrt[3]], i BesselJ[0, 3] Erfi[2 Sqrt[2]], i BesselJ[0, 4] Erfi[Sqrt[15]],
  i BesselJ[0, 5] Erfi[2 Sqrt[6]], i BesselJ[0, 6] Erfi[Sqrt[35]], i BesselJ[0, 7] Erfi[4 Sqrt[3]],
  i BesselJ[0, 8] Erfi[3 Sqrt[7]], i BesselJ[0, 9] Erfi[4 Sqrt[5]], i BesselJ[0, 10] Erfi[3 Sqrt[11]]}}
```

```
Clear[j]
j[x_] := Erf[Sqrt[1 - x^2]] BesselJ[0, x]
Timing[Table[j[i], {i, 10}]]
```

```
{0.25 Second,
 {0, i BesselJ[0, 2] Erfi[Sqrt[3]], i BesselJ[0, 3] Erfi[2 Sqrt[2]], i BesselJ[0, 4] Erfi[Sqrt[15]],
  i BesselJ[0, 5] Erfi[2 Sqrt[6]], i BesselJ[0, 6] Erfi[Sqrt[35]], i BesselJ[0, 7] Erfi[4 Sqrt[3]],
  i BesselJ[0, 8] Erfi[3 Sqrt[7]], i BesselJ[0, 9] Erfi[4 Sqrt[5]], i BesselJ[0, 10] Erfi[3 Sqrt[11]]}}
```