Chapter
On Functional Style

Text for book on Programming in Mathematica
ed. Troels Petersen

Sha Xin Wei
Stanford University
1 August 1995
27 December 1995
1 February 1997

Prologue

Mathematica provides powerful facilities to create and transform structures by composing functions. These structure operators can radically simplify computations that would be tedious to write and read in a procedural programming language like C++ or Pascal. In this chapter, we'll see how we can turn functional programming to our advantage. We will motivate the use of these functional programming features with examples from mathematics.

Structures

Before we start composing functions, let's examine the structures on which all *Mathematica* functions operate. Let's start with a vector `v` and a simple tree of three layers, `m`.

```
v = {1, 2, 3};
m = {1, {21, 22, {231, 232}}, {31, {321, 322}, 33}};
```

■ Structure display functions

There are a few structure display functions that are convenient to use for inspecting some complex structures: `FullForm, TreeForm, Short, Shallow`. Whereas `FullForm` presents the expression in Mathemaica's standard one-dimensional syntax, `TreeForm` draws a crude representation of the structure's expression tree:

```
FullForm[m]
```

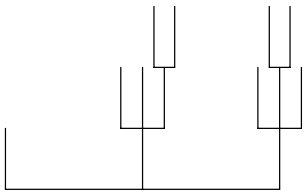
```
List[1, List[21, 22, List[231, 232]], List[31, List[321, 322], 33]]
```

```
TreeForm[m]
```

```
List[1, |                          , |                        ]
        List[21, 22, |             ]  List[31, |             , 33]
                     List[231, 232]             List[321, 322]
```

In version 2.2, there is a standard package `DiscreteMath`Tree`` which we can use to draw expresison trees:

```
Needs["DiscreteMath`Tree`"];
ExprPlot[m]
```



```
-Graphics-
```

Armed with these overviews of structures, we can pick out elements; by convention, the zeroth element of a tree is the Head. [Wolfram]

```
m[[0]]
```

```
List
```

```
m[[1]]
```

```
1
```

```
m[[2]]
```

```
{21, 22, {231, 232}}
```

```
m[[2,1]]
```

```
21
```

```
m[[2,3]]
```

```
{231, 232}
```

```
m[[2,3,2]]
```

```
232
```

```
m[[3]]
```

```
{31, {321, 322}, 33}
```

Many functions in *Mathematica* can work on one or more levels (layers) of an expression tree. Levels are typically specified by a pair of integers {m,n}. Often n means {1,n}, {n} means {n,n}.

```
m[[{2,3}]]
```

```
{{21, 22, {231, 232}}, {31, {321, 322}, 33}}
```

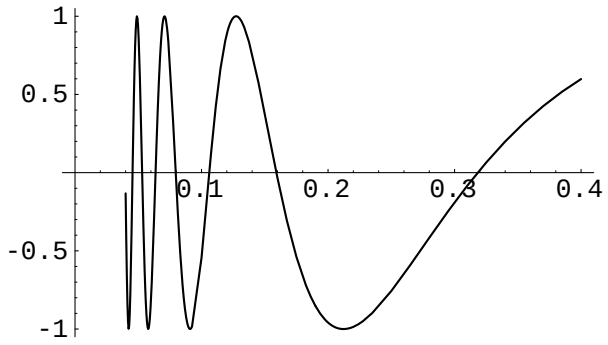
```
m[[2, {1,2}]]
```

```
{21, 22}
```

■ A more complex structure

For larger or more complex structures, it's often convenient to abbreviate the representation drawn by `TreeForm` using `Short` or `Shallow`. For example, `Plot` returns a `Graphics[]` object

```
curve = Plot[Sin[1/x], {x, 0.04, 0.4}];
```



which is actually a list of two lists, most easily seen by

```
Shallow[TreeForm[curve]]
```

```
Graphics[|                , |                ]
         List[|          ]  List[|           ]
              Skeleton[1]         Skeleton[22]
```

Since the curve is a one-dimensional graphics object, we know (or suspect) that there's a `Line` somewhere in its expression.

```
Position[curve, Line]
```

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

The 0 means that `Line` appears as the head of part {1,1,1}. We can verify this by extracting the part at position {1,1,1}

```
Short[
    curve[[1,1,1]]
]
```

```
Line[{{0.04, -0.132352}, <<138>>, {0.4, 0.598472}}]
```

```
Short[
    curve[[1,1,1,1,140]]
]
```

```
{0.4, 0.598472}
```

```
curve[[1,1,1,1,140,2]]
```

```
0.598472
```

Functions of Structures

Mathematica's structural transformation operators, such as `List, Flatten, Partition`, `Transpose`, `Join`, `First`, `Rest`, and `Drop`, provide much of the power traditionally associated with *LISP*, but in the context of rich mathematical structures, these functions can be even more expressive. `Flatten` is probably the most popular structure manipulation function in *Mathematica* after `List`. They are all quite simple, so I'll refer the reader to Wolfram for their definition, but we'll use them in later examples. You may also read about these functions in *Mathematica*'s online Function Browser.

Functions of Functions

A good way to understand how a higher-order function works is to use undefined identifiers as a dummy arguments. We use this idea to understand how *Mathematica's* higher-order functions apply a function to a structure. [Gaylord 61-64] The basic ones are

```
Apply, Map, MapAt, Thread, Through, Fold, Nest, FoldList, NestList
```

The simplest higher order function is `Map`, which applies a function to each element of a list.

```
Map[ f, v]
```

```
{f[1], f[2], f[3]}
```

We can apply a function to one level of an expression tree

```
Map[f, m, {3}]
```

```
{1, {21, 22, {f[231], f[232]}}, {31, {f[321], f[322]}, 33}}
```

Note that this spans two branches. In the next example, we map f repeatedly over the first two levels

```
Map[f, m, 2]
```

```
{f[1], f[{f[21], f[22], f[{231, 232}]}],
  f[{f[31], f[{321, 322}], f[33]}]}
```

```
Map[f, m, {2,3}]
```

```
{f[1], f[{f[21], f[22], f[{f[231], f[232]}]}],
  f[{f[31], f[{f[321], f[322]}], f[33]}]}
```

Note that `Map[f, m, {0}]` is different from `Apply` in that it doesn't replace the head `List`

```
Map[f, m, {0}]
```

```
f[{1, {21, 22, {231, 232}}, {31, {321, 322}, 33}}]
```

To apply a function at a specific position. Let's see what's at position {2,3} of m:

```
m[[2,3]]
```

```
{231, 232}
```

We can map a function f at element {2,3} of m. Notice the entire structure is returned

```
MapAt[f, m, {2,3}]
```

```
{1, {21, 22, f[{231, 232}]}, {31, {321, 322}, 33}}
```

How is this useful? Take the example `curve` from section []. *Mathematica*'s built-in Plot function adaptively estimates functions more often where it becomes more oscillatory. We can extract those points for own purposes using

```
curve[[1,1,1,1]]
```

Other simple functions which may be overlooked on a first reading of the *Mathematica* reference are `Through` and `Thread` Operate, Through, Thread, MapThread, MapIndexed, Scan

■ Operate

`Operate` is a way to define higher-order functions -- functions on functions.

```
Operate[p,f[y]]
```

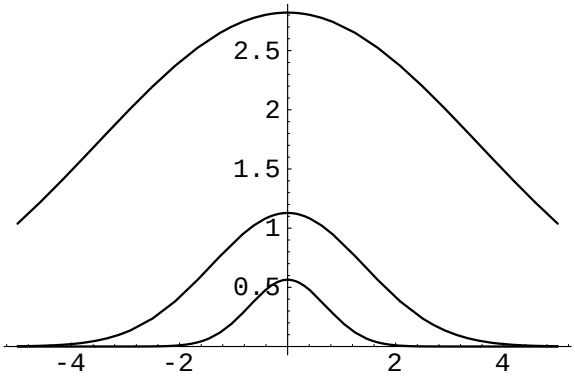
```
p[f][y]
```

It's natural to use `Operate` to define a smoothing operator on real integrable functions. We define the infinitely differentiable function `h0`

```
Clear[h0];
h0[y_] := h0[y,1];
h0[y_,a_,b_:1] := a Pi^(-1/2) b E^(-(y/a)^2);

Integrate[ h0[x], {x, -Infinity, Infinity}]
```

```
1
```

```
Plot[ {h0[x,1], h0[x,2], h0[x,5]}, {x, -5,5}]
```




```
-Graphics-
```

```
Clear[Convolve];
Convolve[f_] := Integrate[ h0[x] * f[x - #], {x, -Infinity, Infinity}]&

Smooth[f_,x_] := Operate[ Convolve, f[x] ]
```

```
Smooth[g,z]

          g[x - z]
Integrate[--------, {x, -Infinity, Infinity}]
              2
             x
            E
---------------------------------------------
                  Sqrt[Pi]
```

[Gilbarg & Trudinger, EPDE's Distribution theory]

■ MapIndexed

In a sense, `MapIndexed` is a crutch to help a function navigate through a complex expression tree. Take a simple example:

```
MapIndexed[f, {a,{b1,b2}}]

{f[a, {1}], f[{b1, b2}, {2}]}
```

The function f, which should be a function of two variables, is called on each item in the list, and told the index of the item. To be concrete, let's define a function `report` that simply prints the second variable, i.e. the index of the part of the expression being fed to the function's first variable.

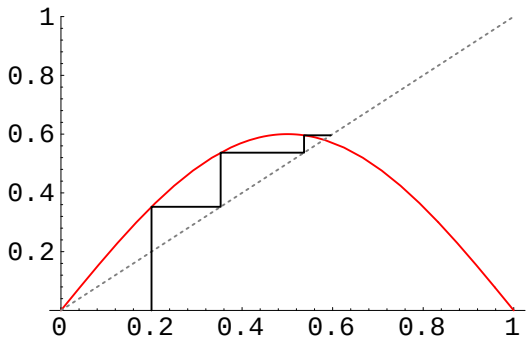
```
report[_,u_] := (Print[u];u);
report[_,u_] := u

MapIndexed[report, {a,{b1,b2}}]

{1}
{2}

{{1}, {2}}

MapIndexed[report, {a,{b1,b2}}, 2]

{1}
{2, 1}
{2, 2}
{2}

{{1}, {2}}
```

■ Iterated functions on an interval

One of the loveliest theorems of elementary topology is that every continuous selfmap of a closed ball must have a fixed point.[Milnor p. 14] In the case of dimension 1, for example,
this means that for any continuous function f : [0,1] -> [0,1], there must be some point x in (0,1) such that f[x] = x.

Often, we wish to apply a function repeatedly -- $T^n(A)$ -- for example, on the study of dynamical systems or lattice physics. `Nest` and `NestList` are tailored for such purposes. More precisely, we take a value f[x], and plug it as an argument back into f: f[f[x]].

We can illustrate the iterates x, f[x], f^2[x], f^3[x] graphically by moving from a point on the graph (x, f[x]) to the point on the graph (x,x), like this:



How would we define a function to produce such a graph? We would like to start with (x,0), and then draw the the staircase of points (x,x), (x, f[x]), (f[x], f[x]), (f[x], f^2[x]), (f^2[x], f^2[x]), (f^2[x], f^3[x]) ,..between y = f[x] and the line y = x. Notice that if we simply take the sequence of values without concerning their role as abscissa or ordinate, we see that each value is repeated four times, and that the distinct values are simply the iterates {f^n[x]}.

■ NestList

`NestList` accumulates the results of iterating a function `f` on an argument `x`:

```
iterates = NestList[f,x,3]
```

```
{x, f[x], f[f[x]], f[f[f[x]]]}
```

We can easily duplicate a list by List, as a pure function, whose elements are repetitions of tha argument:

```
ps = Flatten[
    {#,#,#,#}& /@ iterates
    ]
```

```
{x, x, x, x, f[x], f[x], f[x], f[x], f[f[x]], f[f[x]], f[f[x]], f[f[x]],
  f[f[f[x]]], f[f[f[x]]], f[f[f[x]]], f[f[f[x]]]}
```

We need to account for the first point, (x,0), which is special, and drop the last value so that we have an even number of values, using the First and Rest and Drop list functions.

```
ps2 = Join[
    {First[ps], 0 }, Rest[Drop[ps, -1]]
]
```

```
{x, 0, x, x, x, f[x], f[x], f[x], f[x], f[f[x]], f[f[x]], f[f[x]],
  f[f[x]], f[f[f[x]]], f[f[f[x]]], f[f[f[x]]]}
```

and finally group the sequence of values into ordered pairs.

```
Partition[ps2,2]
```

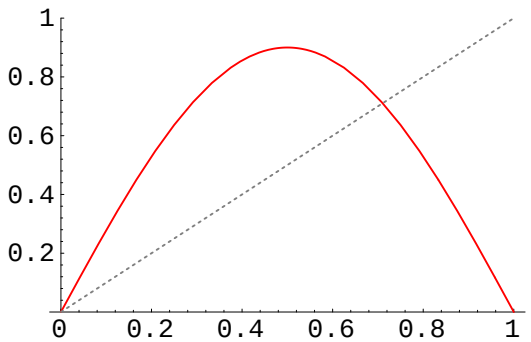
```
{{x, 0}, {x, x}, {x, f[x]}, {f[x], f[x]}, {f[x], f[f[x]]},
  {f[f[x]], f[f[x]]}, {f[f[x]], f[f[f[x]]]}, {f[f[f[x]]], f[f[f[x]]]}}
```

Finally, we string together our sequence of points in a `Line` Graphics object and plot it. Summarizing all these steps in a single program, we have:

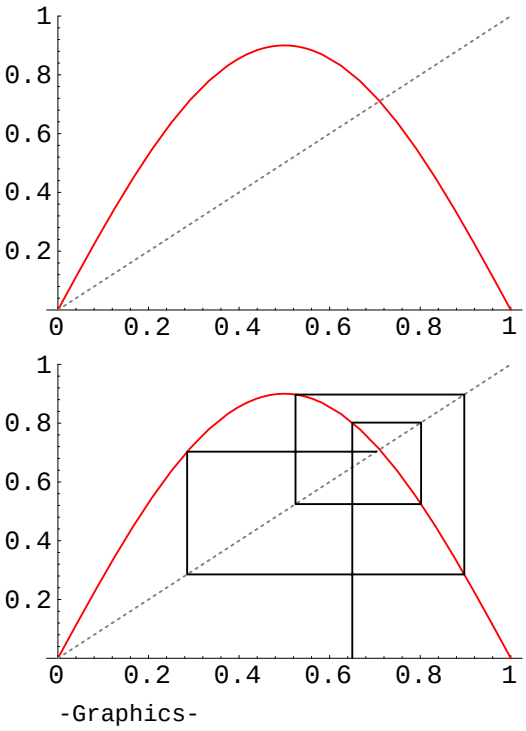
```
Clear[Staircase];
Staircase[f_,x0_,steps_] := Module[{iterates,pointlist,g1},

    iterates = NestList[f,x0,steps];
    pointlist = Flatten[
        {#,#,#,#}& /@ iterates
        ];
    pointlist = Join[
        {First[pointlist], 0 }, Rest[Drop[pointlist, -1]]
    ];
    pointlist = Partition[pointlist,2];
    g1 = Plot[{f[x], x}, {x,0,1},
        PlotRange->{0,1},
        PlotStyle->{{RGBColor[1,0,0]},{GrayLevel[0.5],Dashing[{0.005,0.01}]}}
        ];
    Show[g1,Graphics[Line[pointlist]]]
]
```

(See chapter XXX for a description of *Mathematica*'s plotting functions.) Let's try out our `Staircase` function on a particular automorphism of the unit interval.

```
f1[x_] := 0.9 Sin[x Pi]

Plot[{f1[x], x}, {x,0,1},
    PlotRange->{0,1},
    PlotStyle->{{RGBColor[1,0,0]},{GrayLevel[0.5],Dashing[{0.005,0.01}]}}
    ];
```



```
Staircase[f1, 0.65,5]
```




```
-Graphics-
```

■ Sounding out logistic map

A function related to `NestList[f, expr, n]` is `FoldList[f, x, {a, b, ...}]`, which differs mainly in feeding a second argument to f[x,_]. The second argument replaces the index n used by `NestList, so that`

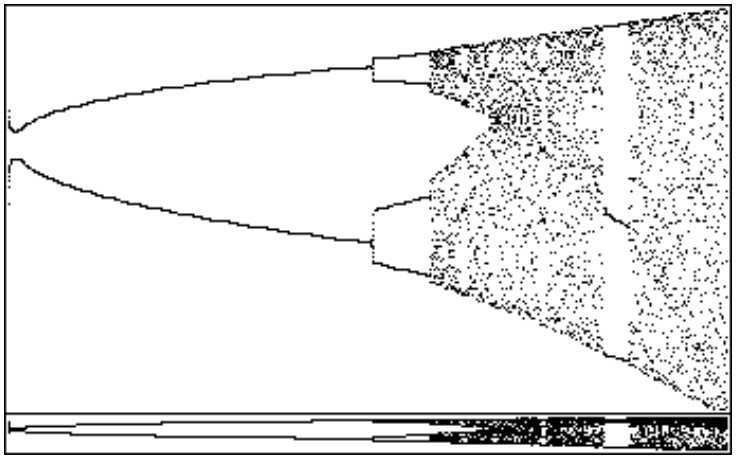
`NestList[f, x, n] == FoldList[f[#1]&, x, Range[n]].`

```
NestList[f, x, 3] == FoldList[f[#1]&, x, Range[3]]
```

```
True
```

MathSource has an elegant example of an application of `FoldList` to sounding out the popular logistic map.

(reprinted from ChaosSound.ma from MathSource.WRI.com.)
Click the speaker icon in the cell bracket to play a sound once. Click the sound graphic to play the sound in a loop. Click outside the graphic or press any key to stop the loop. To control the volume, use the keys next to the Power key on your keyboard (NeXT), or the volume control setting in your Control Panel (Macintosh).



Bifurcation diagram of the logistic map

```
logistic[n_Integer] :=
        Module[ {f, t, x},
            f = Compile[{x, t}, Evaluate[(3 + t/n) x (1 - x)]];
            FoldList[f, 0.223, Range[n]]
        ]

ListPlay[logistic[8000], SampleRate -> 2000]
```

(Multi-)Linear Algebra

Through, Thread, Dot, Transpose, Inner, Outer, Scan,Composition,

■ Through

It can be convenient to use `Through` to evaluate vector-valued functions on a common parameter. For example, if x and y are functions of t, then the vector function {x,y} [t] can be defined as

```
Through[{x,y}[t]]
```

```
{x[t], y[t]}
```

As an example, take a system of ordinary differential equations, which yields two solutions x[t] and y[t]:

```
solution = DSolve[ {x'[t] == y[t], y'[t] == x[t],
    x[0] == 1, y[0] == 0}, {x,y}, {t}];
```

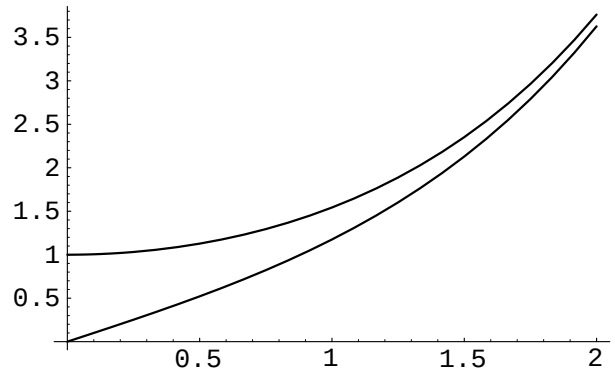
We subsitute the solution into `{x,y}[t]` to get:

```
Through[{x,y}[t]] /. solution
```

$$\left\{\left\{\frac{1}{2\,E^{t}}+\frac{E^{t}}{2},\ \frac{-1}{2\,E^{t}}+\frac{E^{t}}{2}\right\}\right\}$$

Let's plot the result. (We must use `Evaluate` in order to perform all the substitutions for the actual solutions.)

```
Plot[
    Evaluate[Through[{x,y}[t]] /. solution],
    {t, 0, 2}
    ]
```




```
-Graphics-
```

■ Thread

Thread is dual to the `Through` function which we saw earlier. Whereas `Through` applies the componenets of a vector-valued function to a scalar variable, `Thread` applies a scalar function to the components of a vector variable.

```
Thread[f[{x,y}]]
```

```
{f[x], f[y]}
```

Remark: We can simulate this in some cases by declaring a particular function f to be Listable [cf Wolfram p.], but (1) this is dangerous, and (2) you must declare this for each function f in advance.

■ Inner products

The `Dot` product of two vectors

```
Dot[ {a,b,c}, {x,y,z}]
```

```
a x + b y + c z
```

is generalized by `Inner`, replacing `Times` and `Plus` by functions of your choice:

```
Inner[f, {a,b,c}, {x,y,z}, g]
```

```
g[f[a, x], f[b, y], f[c, z]]
```

(In linear algebra, the equivalent of the dot product in general vector spaces is often called the inner product.) Letting `f = Times`, and `g = Plus`, we get the usual `Dot` product back.

```
Inner[Times, {a,b,c}, {x,y,z}, Plus]
```

```
a x + b y + c z
```

■ Example: Prime factorization

By replacing the binary operators `Times` and `Plus`, we obtain a way to recover integers from their prime factorizations. Take for example the test

```
Inner[Power, {a,b,c}, {x,y,z}, Times]
```

$$a^x\ b^y\ c^z$$

Take the prime factorization of 1666

```
factors = FactorInteger[16184]
```

```
{{2, 3}, {7, 1}, {17, 2}}
```

```
Apply[f, factors,2]
```

```
{f[2, 3], f[7, 1], f[17, 2]}
```

```
Times @@ ( Apply[Power, factors, 2])
```

```
16184
```

for example. We can also recover the original integer using

thread and transpose and lin algebra

```
{primes, exponents} = Transpose[  FactorInteger[16184]  ]
```

```
{{2, 7, 17}, {3, 1, 2}}
```

```
Inner[Power, primes, exponents, Times]
```

```
16184
```

As a footnote, we can also selectively freeze the evaluation of the `Power` function in order to display the prime factorization:

```
Inner[ HoldForm[Power[#1,#2]]&, primes, exponents, Times]
```

$$2^3\ 7^1\ 17^2$$

```
ReleaseHold[%]
```

```
16184
```

■ Example: Projecting string vectors

```
sv = {"a","b", "c"};
```

```
(m3 = IdentityMatrix[3]) // MatrixForm
```

```
1   0   0
0   1   0
0   0   1
```

```
m3
```

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

```
n3 = {{1, 0, 1}, {0, 1, 1}, {0, 0, 1}};
```

```
Clear[sproduct];
sproduct[n_Integer,s_String] := sproduct[{n,s}];
sproduct[{n_Integer,s_String}] := StringJoin @@ Array[s&, n]
```

```
Inner[sproduct, n3, sv, List]
```

```
{{a, , c}, {, b, c}, {, , c}}
```

```
Inner[sproduct, n3, sv, StringJoin]
```

```
{ac, bc, c}
```

Exercise? Define a map that will concatenate specified components of a string vector.

■ Outer product

In the simplest case, the outer product of two vectors is a matrix representing the Cartesian product, or cross product, of the two lists.

```
a = {a1,a2};
b = {b1,b2,b3};
Outer[List, a,b] // ColumnForm
```

```
{{a1, b1}, {a1, b2}, {a1, b3}}
{{a2, b1}, {a2, b2}, {a2, b3}}
```

But we can generalize this to replace the List in the first argument by any function:

```
Outer[f, a,b]  // ColumnForm
```

```
{f[a1, b1], f[a1, b2], f[a1, b3]}
{f[a2, b1], f[a2, b2], f[a2, b3]}
```

```
a = {a1,a2, a3};
c = {c1, {c21,c22}};
```

The outer product of a singleton `{a1}` with a list `c` is isomorphic to the list:

```
Outer[f, {a1}, c]
```

```
{{f[a1, c1], {f[a1, c21], f[a1, c22]}}}
```

You can think of the outer product of two lists as replacing each element in the first list by that element crossed with a copy of the second list.

```
Outer[f, a, c] // TableForm

            f[a1, c21]
f[a1, c1]   f[a1, c22]

            f[a2, c21]
f[a2, c1]   f[a2, c22]
```

Note that the outer product is not commutative! Usually `Outer[f, a, c]` will not even have the same shape as `Outer[f, c, a]`.

```
Outer[f, c, a] // TableForm

f[c1, a1]    f[c1, a2]    f[c1, a3]

f[c21, a1]   f[c22, a1]
f[c21, a2]   f[c22, a2]
f[c21, a3]   f[c22, a3]
```

■ Example: Multivariate Calculus

Using the Outer, we can easily define a differential operator on functions.

Given a function f : R^m -> R^n, f(x) = {f_1(x), f_2(x), ..., f_n(x)} , where x = {x_1, x_2, ..., x_m}, the linear map best approximating it at x is given by the Hessian of f's partial derivatives.

$$\frac{\delta f_i}{\delta x_j}(x)$$

```
D[f[x1,x2],x2]
```

$$f^{(0,1)}[x1, x2]$$

We can succinctly define this operator by

```
Clear[Hessian];
Hessian[f_,x_]:= Outer[D, f[x],x]
```

Let's take an example of a function h: R^2 -> R^3.

```
Clear[h];
h[{x1_,x2_}] := {h1[x1,x2],h2[x1,x2],h3[x1,x2]};

( hessh = Hessian[h,{x1,x2}] )  // MatrixForm
```

$$\begin{matrix} h1^{(1,0)}[x1, x2] & h1^{(0,1)}[x1, x2] \\ h2^{(1,0)}[x1, x2] & h2^{(0,1)}[x1, x2] \\ h3^{(1,0)}[x1, x2] & h3^{(0,1)}[x1, x2] \end{matrix}$$

```
Dimensions[hessh]
```

```
{3, 2}
```

```
Transpose[hessh]  . hessh
```

```
{{h1^(1,0)[x1, x2]^2 + h2^(1,0)[x1, x2]^2 +
    h3^(1,0)[x1, x2]^2, h1^(0,1)[x1, x2] h1^(1,0)[x1, x2] +
    h2^(0,1)[x1, x2] h2^(1,0)[x1, x2] +
    h3^(0,1)[x1, x2] h3^(1,0)[x1, x2]},
  {h1^(0,1)[x1, x2] h1^(1,0)[x1, x2] +
    h2^(0,1)[x1, x2] h2^(1,0)[x1, x2] +
    h3^(0,1)[x1, x2] h3^(1,0)[x1, x2],
   h1^(0,1)[x1, x2]^2 + h2^(0,1)[x1, x2]^2 +
    h3^(0,1)[x1, x2]^2}}
```

```
Clear[XXJacobian];
XXJacobian[f_,x_]:= Det[
               Transpose[Hessian[f,x]] . Hessian[f,x]
               ]
```

```
XXJacobian[h,{x1,x2}]
```

```
h2^(0,1)[x1, x2]^2 h1^(1,0)[x1, x2]^2 + h3^(0,1)[x1, x2]^2 h1^(1,0)[x1, x2]^2 -
  2 h1^(0,1)[x1, x2] h2^(0,1)[x1, x2] h1^(1,0)[x1, x2] h2^(1,0)[x1, x2] +
  h1^(0,1)[x1, x2]^2 h2^(1,0)[x1, x2]^2 + h3^(0,1)[x1, x2]^2 h2^(1,0)[x1, x2]^2 -
  2 h1^(0,1)[x1, x2] h3^(0,1)[x1, x2] h1^(1,0)[x1, x2] h3^(1,0)[x1, x2] -
  2 h2^(0,1)[x1, x2] h3^(0,1)[x1, x2] h2^(1,0)[x1, x2] h3^(1,0)[x1, x2] +
  h1^(0,1)[x1, x2]^2 h3^(1,0)[x1, x2]^2 + h2^(0,1)[x1, x2]^2 h3^(1,0)[x1, x2]^2
```

Examples

■ Bitmap processing

■ RasterArray from bitmap 1 (letterF)

Let's first start with a small RasterArray bitmap defined as `LetterF` in the next closed cell. (It was created by pasting a small chip from the *Mathematica* icon into the Front End.)

We import a bitmap of the letter "F"

The next cell is closed to hide the ugly array. Execute it to enter the bitmap as `LetterF`.

```
Show[LetterF]
```

```
-Graphics-
```

We explore the structure of this 7 x 9 matrix.

```
Shallow[
    TreeForm[LetterF],6
]
```

```
Graphics[|                                                          ]
         List[|                                                      ]
              RasterArray[|                 , |                    ]
                          List[|           ] List[|           ]
                               Skeleton[12]       Skeleton[2]
```

From Wolfram, we know that *Mathematica* stores bitmaps as `RasterArray` of `RGBColor`'s, so let's see where it appears in the structure:

```
Position[LetterF, RasterArray]
```

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

Since the head `RasterArray` appears at position{1,1,0}, we know that the bitmap itself must be in position {1,1,1}:

```
Short[
    LetterF[[1,1,1]]
]
```

```
{{RGBColor[0.2863, 0., 0.], <<9>>, RGBColor[0.5725, 0., 0.]}, <<11>>}
```

```
LetterF[[1,1,1]] [[1,1]]
```

```
RGBColor[0.2863, 0., 0.]
```

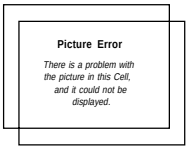
```
Dimensions[LetterF[[1,1,1]]]
```

```
{12, 11}
```

```
Show[
    GraphicsArray[
        {{LetterF},
        {MapAt[Transpose,LetterF,{1,1,1}] , MapAt[Reverse,LetterF,{1,1,1}] }
        }
    ]
];
```

■ RasterArray from a bitmap 2

Here's the original bitmap pasted into the *Mathematica* Notebook Front End.



Using Graph | Convert to InputForm, we transform the bitmap into a Graphics object named `bitmap2`.

First we get a rough view of the structure using `TreeForm`. Since the raster image has many points, we use `Shallow` to abbreviate the report.

```
Shallow[TreeForm[bitmap2],6]
```

```
Graphics[|                                                          ,
         List[|                                                      ]
              RasterArray[|                 , |                    ]
                          List[|           ] List[|           ]
                               Skeleton[71]       Skeleton[2]

  |                                                                  ]
  List[|                          , |                               ]
       Rule[AspectRatio, 0.78889]  Rule[PlotRange, |                 ]
                                                   List[|           ]
                                                        Skeleton[2]

Dimensions[
    bitmap2[[1,1,1]]
]
```

```
{71, 90}
```

Let's apply the same operator to bitmap2 that we applied to LetterF

```
Show[
    GraphicsArray[
        {{bitmap2},
        {MapAt[Transpose,bitmap2,{1,1,1}] , MapAt[Reverse,bitmap2,{1,1,1}] }
        }
    ]
];
```

■ Define functions which acts on points

```
Clear[red];
red[RGBColor[r_,g_,b_]] := RGBColor[r,0,0];
Attributes[red] := {Listable};
Clear[redlevel];
redlevel[RGBColor[r_,g_,b_]] := GrayLevel[r];
Attributes[redlevel] := {Listable};

Clear[green];
green[RGBColor[r_,g_,b_]] := RGBColor[0,g,0];
Attributes[green] := {Listable};
Clear[greenlevel];
greenlevel[RGBColor[r_,g_,b_]] := GrayLevel[g];
Attributes[greenlevel] := {Listable}

Clear[blue];
blue[RGBColor[r_,g_,b_]] := RGBColor[0,0,b];
Attributes[blue] := {Listable};

Clear[bluelevel];
bluelevel[RGBColor[r_,g_,b_]] := GrayLevel[b];
Attributes[bluelevel] := {Listable};

Clear[gray];
gray[RGBColor[r_,g_,b_],maxc_] := GrayLevel[Sqrt[r+g+b/maxc]];
Attributes[gray] := {Listable};
Clear[ceil];
ceil[RGBColor[r_,g_,b_],curmax_] := Module[{newc},
    newc = r^2+b^2+g^2;
    If[ curmax < newc , curmax = newc , curmax];
];
Attributes[ceil] := {Listable}

Clear[rot];
rot[RGBColor[r_,g_,b_]] := RGBColor[g,b,r];
Attributes[rot] := {Listable};
```

■ Try functions as color separation and rotation

```
Show[GraphicsArray[{
    {bitmap2},
    {MapAt[ red, bitmap2, {1,1,1}], MapAt[ green, bitmap2, {1,1,1}]},
    {MapAt[ blue, bitmap2, {1,1,1}], MapAt[ bluelevel, bitmap2, {1,1,1}]},
    {gr = MapAt[ rot, bitmap2, {1,1,1}], MapAt[ rot, gr, {1,1,1}]}
    }]
]
```

■ Ergodic maps

A transformation T from a topological space to itself is called *ergodic* if for any two subsets A and B of non-zero measure, there is an *n* such that the image $T^n(A)$ meets B. This is one way to formalize the notion of thorough mixing.

It's amusing to try to find ergodic self-maps of subsets of R^2. In the discrete case of a bitmap, this typically degenerates to permuting the pixels in a rectangular arra, but often, each point in the bitmap is represented not by a single scalar but by some more complicated structure (eg.g a 3-vector ofred, green and blue color components).. The approach we'll take is to define a permutation on the *position indices* of the RasterArray, which is simply a permutation of an integer matrix. Then we'll use the permuted matrix to select the bitmap.

To illustrate the idea, we first practice with a simple array `mm`:

```
(mm = {{a1, a2, a3}, {b1, b2, b3}} ) // ColumnForm
```

```
{a1, a2, a3}
{b1, b2, b3}
```

Its indices are given by applying the function `report` which we defined abovd () to justlevel 2 of the array called `mm`.

```
(ijm = MapIndexed[ report, mm, {2}]) // ColumnForm

{{1, 1}, {1, 2}, {1, 3}}
{{2, 1}, {2, 2}, {2, 3}}

(ijm = Array[List, Dimensions[ mm] ] ) // ColumnForm

{{1, 1}, {1, 2}, {1, 3}}
{{2, 1}, {2, 2}, {2, 3}}

ijm == %85

True
```

Here's a simple permutation that exchanges rows.

```
Reverse[ijm]  // ColumnForm

{{2, 1}, {2, 2}, {2, 3}}
{{1, 1}, {1, 2}, {1, 3}}
```

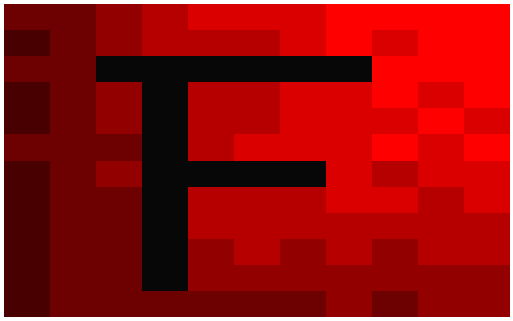
If we "evaluate" mm on this permuted array of indices, we'll get the same permutation of mm's elements.

```
(Apply[mm[[#1,#2]]&, Reverse[ijm], {2}] ) // ColumnForm

{b1, b2, b3}
{a1, a2, a3}
```

Of course, we could have simply Reverse'd mm, but this method allows us to manipulate much more complex structures via this indirection. Let's try apply this indexed permutation idea to a bitmap. We use `ReplacePart,` and `Apply`, in addition to our own permutation T.

Recall the butmap we first used:

```
Show[LetterF]
```



```
-Graphics-
```

which contains bitmap information in an RasterArray:

```
Position[LetterF, RasterArray]

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

We can apply our permutation directly to this array in place using the `ReplacePart` operator. Here's an example of how `ReplacePart` works, placing `dummy` where the array of `RGBColor`'s.

The selection operator is defined as

```
Short[TreeForm[
    ReplacePart[LetterF, dummy, rasterPosition]
    ],5
]
```

```
Graphics[

   |
   List[|                                                                  ]'
        RasterArray[dummy, |                                              ]
                           List[|           , |                          ]
                                List[0., 0.]  List[1., |                  ]
                                                       Rational[7, 9]

  |                                                    ]
  List[|                              , |             ]
       Rule[AspectRatio, Automatic]  Rule[PlotRange, All]
```

Let's collect these ideas into a single program. This way, we need only define functions on R^2, and use them to manipulate the complicated representation of an image in situ. This way, we can re-use the display settings from the original graphic.

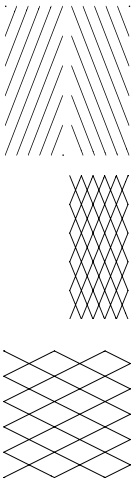
We define a bitmap mixer which uses a function of two variables (`permutor`) to permute an array of indexes into a two-dimensional bitmap. Since the bitmap
is just a structure, we can extract one point out of it by "applying" it to a single index
(i,j). Indeed we can effectively permute it by "applying" it to the entire array.

```
Clear[RegionMix];
RegionMix[original_Graphics, permutor_, steps_:1] :=
Module[{ij,rg,ijs,bitmap,bmaps,T,xij},
    bitmap = original[[1,1,1]];
    ij = Array[List, Take[Dimensions[bitmap], 2] ];
    ijs = NestList[permutor, ij, steps];            (* permute indexes *)
    T[xij_] := Apply[bitmap[[#1,#2]]&, xij, {2}];   (* T "evaluates" bitmap on indexes
xij *)
    bmaps = T /@ ijs;
    Show[ReplacePart[original, #, {1,1,1}]]& /@ bmaps   (* use original's Graphics
options *)
]
```

The last step in the function `RegionMix` just displays the bitmaps. By re-inserting each in place of the bitmap in the original Graphics object `r`, we show the bitmaps using the original's Graphics display settings.

■ The Baker's Map

The baker's map [Petersen] mimics the folding and kneading actions of preparing dough.
In one dimension the baker's map folds a given interval in half, then stretches it back to its original length. In higher dimensions, one iterates this in each dimension. We can represent this graphically like this:



We define a discrete form of the map on a matrix of {i,j} indexes.

```
Clear[FoldMatrix];
FoldMatrix[ij_List] := Module[{height, halfheight, m},
    height = Length[ij];
    halfheight = Quotient[height,2];
    m = MapAt[Reverse,
            Partition[ij,halfheight],
        {2}];
    m = Flatten[
            Transpose[m],1
        ];
    If[EvenQ[height], m, Join[m,{Last[m]} ]  ]
]

(mmm = DiagonalMatrix[Range[4]] ) // MatrixForm
```

```
1   0   0   0
0   2   0   0
0   0   3   0
0   0   0   4
```

```
FoldMatrix[mmm]  // MatrixForm
```

```
1   0   0   0

0   0   0   4

0   2   0   0

0   0   3   0
```

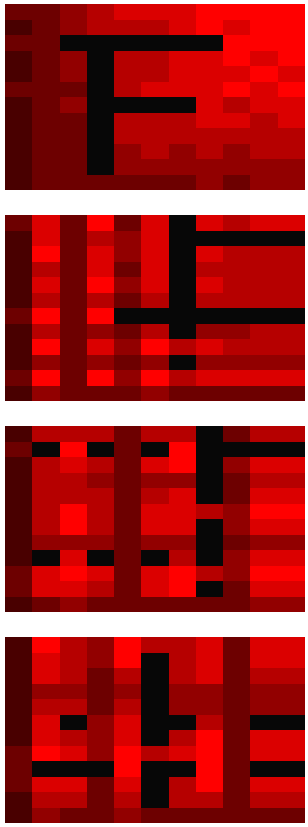
Note that we've defined `FoldMatrix` so that itworks on higher-dimensional arrays, too.

```
( m2 = Outer[List, mmm, {a,b}])  // MatrixForm

1 a   0 a   0 a   0 a
1 b   0 b   0 b   0 b

0 a   2 a   0 a   0 a
0 b   2 b   0 b   0 b

0 a   0 a   3 a   0 a
0 b   0 b   3 b   0 b

0 a   0 a   0 a   4 a
0 b   0 b   0 b   4 b

FoldMatrix [ m2]  // MatrixForm

1 a   0 a   0 a   0 a
1 b   0 b   0 b   0 b

0 a   0 a   0 a   4 a
0 b   0 b   0 b   4 b

0 a   2 a   0 a   0 a
0 b   2 b   0 b   0 b

0 a   0 a   3 a   0 a
0 b   0 b   3 b   0 b
```

Now, the 2D baker's map folds the rectangle in half and stretches it out to its original size in each dimension by turn. This is simply represented by a composition of the `FoldMatrix` and `Transpose` functions, using *Mathematica*'s Comp operator:

```
baker = Composition[Transpose, FoldMatrix, Transpose, FoldMatrix ];

baker[m0]

Transpose[FoldMatrix[Transpose[FoldMatrix[m0]]]]

(baker[mmm] ) //MatrixForm

1   0   0   0
0   4   0   0
0   0   2   0
0   0   0   3

baker[m2] // MatrixForm

1 a   0 a   0 a   0 a
1 b   0 b   0 b   0 b

0 a   4 a   0 a   0 a
0 b   4 b   0 b   0 b

0 a   0 a   2 a   0 a
0 b   0 b   2 b   0 b

0 a   0 a   0 a   3 a
0 b   0 b   0 b   3 b
```

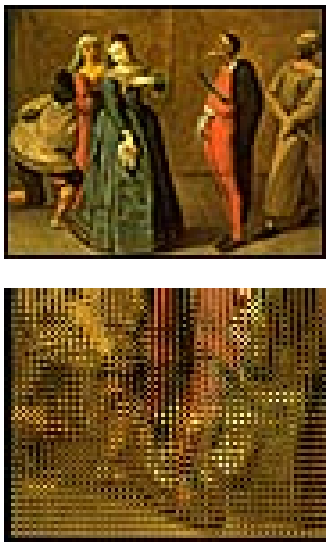
```
RegionMix[LetterF, baker, 3]
```

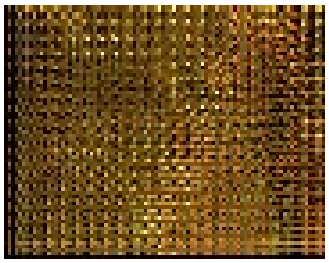


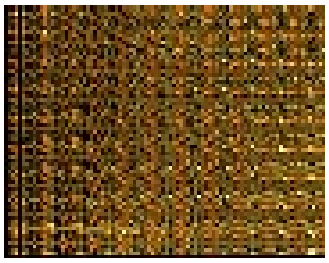
```
{-Graphics-, -Graphics-, -Graphics-, -Graphics-}
```

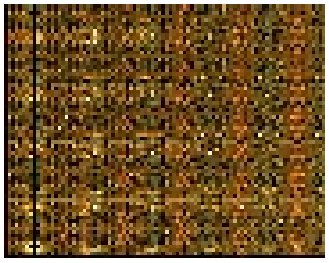
Now let's apply this `baker` map to the `bitmap2` we used above:

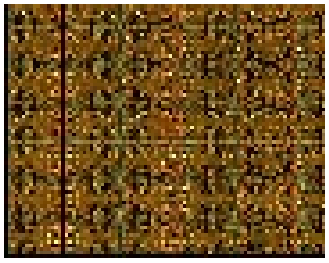
```
RegionMix[bitmap2, baker, 5]
```











■ *Exercise*: Irrational slide on a torus

Now let's define a map that slides points along lines of a fixed irrational slope. To make the map into a self-map of a rectangle, we identify the edges of a rectangle as usual to make a torus. (Why should the slope by irrational? What happens if the slope is k/m where k and m are integers?)

```
Clear[Round1];
Round1[x_] := If[(t = Round[x])==0, 1, t];
SetAttributes[Round1,Listable];

Clear[T];
T[{x_,y_},theta_:Sqrt[2]] := T[{x,y},theta,{10,10}];
T[{x_,y_},theta_, {xspan_, yspan_}] := Round1[N[Mod[
    { x + Cos[theta], y + Sin[theta]}, {xspan,yspan }
    ]]];

Clear[Ts];
Ts[{x_,y_},modulus_,slope_] := Ts[{x,y},modulus,slope,{10,10}];
Ts[{x_,y_},modulus_, slope_, {xspan_, yspan_}] := Mod[
    N[{x + modulus, y + modulus slope}], {xspan,yspan}
    ];
```

We define the rectangle of indices from the dimensions of the bitmap:

```
( LetterFindices = Array[List, Dimensions[LetterF[[1,1,1]]]  ]) // ColumnForm

{{1, 1}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6}, {1, 7}, {1, 8},

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

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

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

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

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

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

  {7, 9}}

Dimensions[LetterFindices]

{7, 9, 2}

LetterFSize = Take[Dimensions[LetterFindices],2]

{7, 9}
```

In other words, a 7 by 9 array of two-dimensional indices. Now we can customize our irrational slide map to the dimensions of this bitmap:

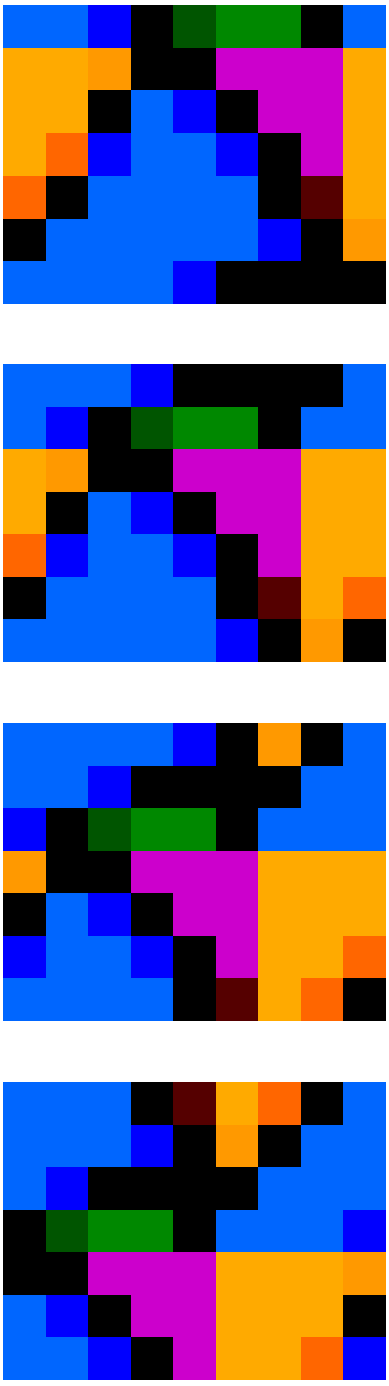
```
Clear[TE];
TE[i_,j_,bitmapSize_] := Ts[{i,j},5,1/Sqrt[2], bitmapSize];
```

TA and TE are the maps that we actually use to permute the indices.

```
Clear[TA1];
TA1[i_,j_] := TE[i,j, LetterFSize];
Apply[TA1, LetterFindices, {2}] // ColumnForm
```

```
{{6., 4.53553}, {6., 5.53553}, {6., 6.53553}, {6., 7.53553},

  {6., 8.53553}, {6., 0.535534}, {6., 1.53553}, {6., 2.53553},

  {6., 3.53553}}
{{0., 4.53553}, {0., 5.53553}, {0., 6.53553}, {0., 7.53553},

  {0., 8.53553}, {0., 0.535534}, {0., 1.53553}, {0., 2.53553},

  {0., 3.53553}}
{{1., 4.53553}, {1., 5.53553}, {1., 6.53553}, {1., 7.53553},

  {1., 8.53553}, {1., 0.535534}, {1., 1.53553}, {1., 2.53553},

  {1., 3.53553}}
{{2., 4.53553}, {2., 5.53553}, {2., 6.53553}, {2., 7.53553},

  {2., 8.53553}, {2., 0.535534}, {2., 1.53553}, {2., 2.53553},

  {2., 3.53553}}
{{3., 4.53553}, {3., 5.53553}, {3., 6.53553}, {3., 7.53553},

  {3., 8.53553}, {3., 0.535534}, {3., 1.53553}, {3., 2.53553},

  {3., 3.53553}}
{{4., 4.53553}, {4., 5.53553}, {4., 6.53553}, {4., 7.53553},

  {4., 8.53553}, {4., 0.535534}, {4., 1.53553}, {4., 2.53553},

  {4., 3.53553}}
{{5., 4.53553}, {5., 5.53553}, {5., 6.53553}, {5., 7.53553},

  {5., 8.53553}, {5., 0.535534}, {5., 1.53553}, {5., 2.53553},

  {5., 3.53553}}
```

```
NestList[ RegionMix[#,TE]&, LetterF, 4];
```



Now let's apply this to a real bitmap `bitmap2:`

```
bitmap2indices = Array[List, Dimensions[bitmap2[[1,1,1]]]  ];
```

```
bitmap2Size = Take[Dimensions[bitmap2indices],2]
```

```
{71, 90}
```

```
Clear[TA2];
TA2[i_,j_] := TE[i,j,bitmap2Size];
```

Let's just doublecheck that this is not the identity map.

```
TA2[1,1]
```

```
{6, 5}
```

```
Date[]
```

```
{1995, 10, 2, 1, 25, 29}
```

```
b2shifts = NestList[ RegionMix[#,TA2]&, bitmap2, 6];
```

```
Date[]
```

```
{1995, 10, 2, 2, 14, 21}
```

Notes

■ Further reading

Gaylord, ,Wellin.

Gilbarg, Trudinger. Partial Differential Equations.

John Milnor. Topology From the Differentiable Viewpoint.

Karl Petersen, Ergodic Theory

S. Wolfram, Third Edition

■ Index

```
SortIndex["PASTE-INDEX_HERE"] // ColumnForm
```

```
SortIndex["TreeForm...1,3,16,18,22
FullForm...1
Plot...3,5,8,11,18
Shallow...3,16,18
Position...3,16,22
Short...3,16,22
Map...4-5,23
MapAt...5,17,19-20
Operate...5-6
MapIndexed...6,21
NestList...7-9
FoldList...9-10
Through...10-11
Thread...11
Dot...11
Inner...11-13
Apply...4,7,10,12,21-22,27
DSolve...10
ReleaseHold...12
HoldForm...12
MatrixForm...13-14,23-25
IdentityMatrix...13
Array...13,22
List...13
StringJoin...13
Outer...13-14
ExprPlot...2
Integrate...5-6
Evaluate...10-11
Module...8,10,19,22-23
Partition...8,23
First...8
Rest...8
Drop...8
Join...8
TableForm...14
Flatten...7
Dimensions...15-16,18,21-22,27
Transpose...12,15,17,19,23-24
Det...15
RasterArray...15-16,18,22
GraphicsArray...17,19-20
RGBColor...8,15,18-20
ColumnForm...13,21,27
Reverse...17,19,21,23
ReplacePart...22
Composition...24"] // ColumnForm
```

```
StringReplace[#,"."->"\t"]& /@ %81
```

```
Apply   4,7,10,12,21-22,27
Array   13,22
ColumnForm   13,21,27
Composition   24
Det   15
Dimensions   15-16,18,21-22,27
Dot   11
Drop   8
DSolve   10
Evaluate   10-11
ExprPlot   2
First   8
Flatten   7
FoldList   9-10
FullForm   1
GraphicsArray   17,19-20
HoldForm   12
IdentityMatrix   13
Inner   11-13
Integrate   5-6
Join   8
List   13
MapAt   5,17,19-20
MapIndexed   6,21
Map   4-5,23
MatrixForm   13-14,23-25
Module   8,10,19,22-23
NestList   7-9
Operate   5-6
Outer   13-14
Partition   8,23
Plot   3,5,8,11,18
Position   3,16,22
RasterArray   15-16,18,22
ReleaseHold   12
ReplacePart   22
Rest   8
Reverse   17,19,21,23
RGBColor   8,15,18-20
Shallow   3,16,18
Short   3,16,22
StringJoin   13
TableForm   14
Thread   11
Through   10-11
Transpose   12,15,17,19,23-24
TreeForm   1,3,16,18,22
```