Two Tricks for the Price of One: Linear Filters and their Transposes

Dan Piponi
Industrial Light & Magic

May 12, 2009

Abstract

Many image processing algorithms exist in pairs: a forward mapping version and a reverse mapping version [9], [10]. In this paper we show that the forward and reverse versions of spatially-varying convolution filters are transpose to each other. We will then show how an implementation of a function that operates linearly on a set of variables may be transformed into an implementation of its transpose function. This gives a mechanical procedure to convert a reverse spatially-varying convolution into a forward one and vice versa. Although this approach is general purpose we focus on one particular type of application: applying this transformation to fast algorithms for reverse convolution based on running sums or summed area tables [6], [3] yielding novel fast algorithms for forward convolution. For many practical applications, such as simulating depth of field and motion blur, the forward convolution can often yield more visually appealing results while the reverse mapping algorithm has traditionally been more straightforward to implement.

1 Introduction

Suppose we have a sequence of values $(g(i))$ for $i = 0,\ldots,n-1$ and that we wish to implement a function to compute the partial sums $s(a,b) = \sum_{i=a}^{b} g(i)$ for any a and b. If we precompute a sequence $(h(i))$ with $h(0) = 0$ and $h(i+1) = h(i) + g(i)$ so that the $h(i)$'s are the *running sums* of the $g(i)$'s then we may compute $s(a,b)$ as $h(b+1) - h(a)$ in time $O(1)$. This is the well known method of running sums and it can be seen that it provides a method to allow rapid discrete convolution[1] of a sequence $(g(i))$ with a 1-dimensional box filter. We may also rapidly evaluate sums of the form

$$\sum_{i=a}^{b}\sum_{j=c}^{d} g(i,j)$$

by using the well known 2D analogue of running sums, 'summed area tables' [3] and this gives a way to convolve rapidly with a 2D box filter. More generally we can

[1]In this paper we will always be referring to the *discrete* convolution rather than its continuous counterpart.

rapidly convolve with a wide variety of filters using similar methods such as Heckbert's repeated convolution [5] or Sun's polygonal smoothing algorithms [8].

Running sum methods may also be used with more general kernels. For example simulated defocus can be approximated by convolution with a kernel whose shape is the same as that as the iris of the simulated camera. The bulk of this convolution can be computed by representing the filter as the sum of 1D filters, one for each row of the rasterization of the iris shape and using running sums for each row.

The above methods may be used to compute not just ordinary convolutions but also *spatially-varying convolutions*. There are two approaches to defining such filters: *reverse* and *forward* approaches corresponding to two ways of viewing ordinary convolution. We will use the symbols $\triangleleft$ and $\triangleright$ to denote these two forms of convolution.

Firstly, consider the usual definition of convolution:

$$(f * g)(x) = \sum_{x'} f(x - x')g(x').$$

The value of $f * g$ at each point x can be interpreted as a weighted average of values of g 'gathered' from points x' in its neighbourhood using $f(x-x')$ as a weighting. Viewed this way it seems natural to define a spatially varying convolution by

$$(f \triangleleft g)(x) = \sum_{x'} f(x, x - x')g(x')$$

where we have now allowed f to depend on x. We can think of $f(x,x')$ defining a different kernel $f(x,\cdot)$ at each point x at wgich we gather.

We can also view convolution in a dual manner. We can interpret $f * g$ at x as the accumulation of values of g 'scattered' from x and weighted by $f(x-x')$. From this perspective it becomes natural to generalise convolution to spatially varying convolution as

$$(f \triangleright g)(x) = \sum_{x'} f(x', x - x')g(x')$$

where the kernel used for scattering, $f(x',.)$, now depends on the point from which we scatter.

We call the former reverse mode spatially varying convolution and the latter forward mode spatially varying convolution. In the case where the kernel doesn't vary spatially the definitions coincide. ([9] and [10] have some discussion about these two modes of operation in the context of image warping and resampling.) Consider convolution with a spatially varying box filter ie. one for which for each value of x, $f(x,x')$ takes the value 1 when x' is inside an axis-aligned box centred on the origin and 0 outside and where the box size is a function of x. It is clear that we may use running sum methods to compute the reverse mode convolution as each summation in the definition is a sum over a box. Note, however, that the forward mode filter is much more complex. As a function of x', $f(x', x-x')$ might not give rise to simple summations over boxes. So we appear to have an asymmetry: we have rapid algorithms for computing the reverse filters but not the forward filters.

In this paper we will first show how to transform a computer program to compute a linear function into a program to compute its transpose. Next will then show the two

types of spatially varying convolution above are transposes of each other. And then we will apply the former to the latter to show how to transform an algorithm for one type of spatially varying convolution into the other. In particular, a fast algorithm for one becomes a fast algorithm for the other.

2 The Transpose of a Computer Program

Consider the following fragment of code in an imperative programming language such as C++: `a = `α`a+`β`b+`$\ldots$`+`δ`d` where $\alpha, \beta, \ldots, \delta$ are constants. (Actually they don't need to be constants, merely independent of the variables that we are considering to be linear. See section 5 and 6) We may represent this in matrix form as the update

$$\begin{pmatrix} a \\ b \\ \vdots \\ d \end{pmatrix} = \begin{pmatrix} \alpha & \beta & \ldots & \delta \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ \vdots \\ d \end{pmatrix}$$

If we consider `a = `αa`+`βb`+`$\ldots$`+`δ`d` as a linear operator acting on $(a, b, \ldots, d)$ then we may derive the transpose linear operator by taking the transpose of the matrix above giving

$$\begin{pmatrix} a \\ b \\ \vdots \\ d \end{pmatrix} = \begin{pmatrix} \alpha & 0 & \ldots & 0 \\ \beta & 1 & & 0 \\ \vdots & & \ddots & \\ \delta & 0 & & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ \vdots \\ d \end{pmatrix}$$

which may be expressed as the code fragment `b = b+`β`a;`$\ldots$`;d = d+`δ`a; a = `α`a .` Suppose more generally that we have a sequence of lines of code $l_1; l_2; \ldots; l_n$ that may each be represented by matrices $L_1, L_2, \ldots, L_n$, then the effect of executing the lines of code in sequence is represented by the matrix $L_n L_{n-1} \ldots L_1$. The transpose of this is simply $L_1^T L_2^T \ldots L_n^T$. So in order to compute the transpose function to that defined by $l_1; l_2; \ldots; l_n$ we must compute the transpose of each line in turn and reverse the order in which they are executed. (By *transpose* we mean here the linear operator given by the transpose matrix in the basis we are using. See [1] for a more rigorous definition set within the context of rendering.)

The following is a table of example code fragments and their transposes:

Note that the transpose of the transpose of a code fragment is equivalent to the original fragment. For example the transpose of `a = b` is `b = a+b; a = 0;` whose transpose in turn is `a = 0; a = a+b` which is equivalent to `a = b`.

There is one slight subtlety when computing transposes. Suppose we have two variables a and b and simply discard b. This may be represented as

$$\begin{pmatrix} a \end{pmatrix} = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

Code	Transpose
`a = 0`	`a = 0`
`a = 2*a`	`a = 2*a`
`a = a+b`	`b = b+a`
`a = b`	`b = a+b; a = 0`
`a = 2*b-3*c`	`b = b+2*a; c = c-3*a; a = 0`
`a = a+b; b = b+2*a`	`a = a+2*b; b = b+a`

Table 1: Some code fragments and their transposes

The transpose is given by

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} a \end{pmatrix}$$

which represents the code fragment `b = 0`. In other words, the transpose of discarding a variable is setting it to zero. Discarding variables is something we typically take for granted when programming, but here we must explicitly note when we are doing this so we can generate the correct transpose, or we must enforce that rule that a variable must be zeroed before being discarded so that in the transpose code it is automatically zeroed.

3 An elementary example

Consider the linear mapping defined by

$$\begin{pmatrix} \texttt{g1} \\ \texttt{g2} \\ \texttt{g3} \\ \texttt{g4} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \texttt{f1} \\ \texttt{f2} \\ \texttt{f3} \\ \texttt{f4} \end{pmatrix}$$

It can be thought of as representing the code

```
g1 = f1+f2;
g2 = f1+f2+f3;
g3 = f2+f4;
g4 = f4;
```

The transpose code fragment is then

```
f1 = 0; f2 = 0; f3 = 0; f4 = 0;
f4 += g4; g4 = 0;
f2 += g3; f4 += g3; g3 = 0;
f1 += g2; f2 += g2; f3 += g2; g2 = 0;
f1 += g1; f2 += g1; g1 = 0;
```

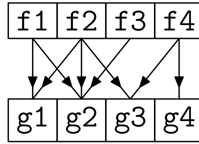

Figure 1: The data flow in the first linear map

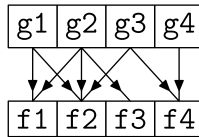

Figure 2: The data flow in the second linear map

And this clearly implements the linear mapping:

$$\begin{pmatrix} \mathtt{f1} \\ \mathtt{f2} \\ \mathtt{f3} \\ \mathtt{f4} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} \mathtt{g1} \\ \mathtt{g2} \\ \mathtt{g3} \\ \mathtt{g4} \end{pmatrix}$$

and we can see the matrix is clearly the transpose of the original matrix.

We can describe these operations diagramatically. The flow of data in the original linear mapping is represented in Figure 1. The transpose mapping represents the transpose linear mapping and is essentially the original diagram inverted. Referring to the implementation of the original function consider the line `g2 = f1+f2+f3`. In can be considered to be 'gathering' values from locations 1, 2 and 3 into location 2. In the transpose code the corresponding line `f1 += g2; f2 += g2; f3 += g2; g2 = 0;` can be seen as 'scattering' the value in the second location into locations 1, 2, 3. Computing the transpose swaps gathering and scattering operations. We will use this duality when we investigate spatially varying convolution by forward and reverse mapping.

4 Transposes and Spatially Varying Convolution

Consider again the two definitions of spatially varying convolution. If we write $m_f(x,y) = f(x,x-y)$ then we get

$$\begin{aligned} (f \triangleleft g)(x) &= \sum_y g(y) m_f(x,y) \\ (f \triangleright g)(x) &= \sum_y g(y) m_{f'}(y,x) \end{aligned}$$

where $f'(x,y) = f(x,-y)$. In other words, if we consider the discrete points x and y to be indices, and $m_f(x,y)$ to be a matrix indexed by x and y, then reverse spatially-varying convolution is precisely the transpose to the corresponding forward spatially-varying convolution with the kernels transformed by $x \to -x$.

5 Transposes and spatially varying convolution

We now have a prescription for computing the transpose of any linear function. Given an implementation of a forward spatially-varying convolution we may convert it to the code for the reverse spatially-varying convolution (with the kernels spatially reversed) simply by reversing the order of all of the linear operations in it and replacing each one with its transpose. In order to illustrate the method we will apply this method to implement running sums to compute spatially varying 1D box filters:

```
 0 void ReverseBoxBlur(int n,float *data,int *radius)
 1 {
 2     float *sum = new float[n+1];
 3
 4     //
 5     // Familiar 'summed area' algorithm...
 6     //
 7    sum[0] = 0;
 8    for (int i = 1; i<=n; ++i)
 9        sum[i] = sum[i-1]+data[i-1];
10
11    for (int i = 0; i<n; ++i)
12        data[i] = sum[i+1+radius[i]]-sum[i-radius[i]];
13
14    for (int i = 0; i<=n; ++i)
15        sum[i] = 0;
16
17    delete[] sum;
18 }
```

There are some important points to note about this code. For clarity of exposition, it assumes that the values stored in ‘radius’ are such that there will be no out-of-bounds array accesses at line 12. We have chosen a filter that is symmetric under $x \to -x$ so that the kernel of the transpose filter doesn’t need to be spatially transformed in this way. It is only linear in the array ‘data’ and we may consider n and the elements of the array ‘radius’ to be constants. At lines 14 and 15 we have explicitly zeroed out the array ’sum’ before it is discarded as noted above. Obviously in production code this may be omitted.

We now transform this code into the following. The lines 7, 9, 12 and 15 perform linear operations and become lines 40, 37/38, 31/32/33 and 28 respectively. The loops at lines 8, 11 and 14 become the loops at lines 36, 30 and 27 respectively.

```
19 void ForwardBoxBlur(int n,float *data,int *radius)
20 {
21     float *sum = new float[n+1];
22
23     //
24     // Less familiar 'render derivative of result'
25     // algorithm...
26     //
27     for (int i = n; i>=0; --i)
28         sum[i] = 0;
29
30     for (int i = n-1; i>=0; --i) {
31         sum[i+1+radius[i]] += data[i];
32         sum[i-radius[i]]   -= data[i];
33         data[i] = 0;
34     }
35
36     for (int i = n; i>=1; --i) {
37         sum[i-1]  += sum[i];
38         data[i-1] += sum[i];
39     }
40     sum[0] = 0;
41
42     delete[] sum;
43 }
```

Note that the final code may be optimised, for example there is some redundancy in lines 36-39.

It is interesting to re-examine the transformed code. Lines 36-39 sum the data in the array `sum`. So lines 30-34 are rendering what is essentially the finite difference of the image. The finite difference of the box filter kernel has two non-zero values, 1 and -1 at the leading and trailing edge. We can see that lines 30-34 perform a convolution with the finite difference of the box filter and lines 36-39 sum this result to convert it to the convolution with the undifferenced box filter. So clearly this code does in fact specify an algorithm for fast forward convolution with a spatially varying box filter.

6 Discussion

We have given a simple example but the principle applies to any linear filter that is expressed as a sequence of basic linear operations. A wide variety of filters may be expressed in this way. Our particular example application of computing the transpose also suggests some interesting rendering algorithms. If we are rendering images into an accumulation buffer it may, for certain types of rendering, be more efficient to instead render the derivative of the image and then perform a final sweep at the end integrating it. For example arbitrary transparent constant colour shapes may be efficiently rendered

additively simply by rendering suitable values along their boundary and integrating at the end. This is the transpose to the more usual technique of filtering scanlines using running sums. More complex primitives such as gouraud shaded triangles may be rendered as second derivatives and then integrated twice. Algorithms such as polygonal smoothing methods also yield fast forward spatially varying convolutions [8].

Like the original running sums algorithms the transpose algorithm may be subject to numerical error. For example each of the terms computed in line 38 above is the sum of many terms accumulated at lines 31, 32 and 37 and hence may accumulate many small rounding errors. One common way to deal with these errors is to copy a technique used for ordinary running sums: work with values that are integers modulo N where N is typically 2^b and b is the computer word size in bits, and rearrange our algorithm to ensure that at each stage we use only integer addition and multiplication. If we know that the final results must lie within the range $0,\ldots,N-1$ then the result of the final computation will be exactly correct even if intermediate results caused overflows outside this range.

The described method of transforming code into its transpose is not novel. However there appear to be very few references to the method in the published literature. Jon Claerbout discusses it with examples in his online book [2]. It is closely related to adjoint mode automatic differentiation [4]. In fact, applying an adjoint mode differentiation algorithm to a linear filter effectively computes the transpose filter [7].

We haven't discussed the 'constants' represented by $\alpha,\beta,\ldots,\delta$ above. In practice these need not be constant but merely independent of the variables that are being linearly transformed. Unfortunately, if the computation of these constants is itself complex, it can happen that reversing the order of the transpose code fragments can interfere with this computation. A well studied example of this is in reverse mode automatic differentiation which is the transpose of forward mode automatic differentiation. In this case the 'non-linear' evaluation must take place once in forward mode followed by another transpose linear pass in reverse mode, similar to the methods used in adjoint mode automatic differentiation. However, for straightforward linear filters these issues are unlikely to arise.

There may be many other applications of this technique in the field of rendering where many of the computations are linear in the input data such as lighting and textures. Any time we have a fast linear algorithm there is the possibility that there exists a fast transpose counterpart.

7 Acknowledgements

Thanks to Christophe Hery, Simon Premoze and JP Lewis for valuable dicussion and thanks to Barnaby Robson for finding errors in an earlier draft.

References

[1] Per H. Christensen. Adjoints and importance in rendering: An overview. 9(3):329–340, July/September 2003.

[2] Jon Claerbout and James L Black. *Basic Earth Imaging*. 2001. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.8870.

[3] Franklin C. Crow. Summed-area tables for texture mapping. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212. ACM Press, 1984.

[4] Andreas Griewank. On Automatic Differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[5] Paul S. Heckbert. Filtering by repeated integration. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 315–321. ACM Press, 1986.

[6] J. Lewis. Fast normalized cross-correlation. Vision Interface., 1995.

[7] Barak Pearlmutter. Personal communication.

[8] Changming Sun. Diamond, hexagon, and general polygonal shaped window smoothing. In Sun C., Talbot H., Ourselin S., and Adriaansen T., editors, *Proc. VIIth Digital Image Computing: Techniques and Applications*, Sydney, 2003.

[9] George Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[10] George Wolberg, H M Sueyllam, M A Ismail, and K M Ahmed. One-dimensional resampling with inverse and forward mapping functions. *Journal of Graphics Tools*, 5(3):11–33, 2000.