A Transform Hyperator in APL

Bob Smith
Sudley Place Software

Originally Written
8 Dec 2018
Updated
21 Jul 2019

Introduction

Hyperators[1] – as defined by John Scholes[4] – are discussed along with their syntax and various examples. A particular example, presented here, is the Transform Hyperator which is an instantiation into a single object of the mathematical concept of a Transform.

Transforms

As the name implies, a **Transform** changes its data from one perspective to another. For example, a Fourier Transform may be thought of as changing data from an equally-spaced quantity that varies over the time domain to an equally-spaced quantity that varies over the frequency domain. However, these “domain”s are quite fluid, as, for example, a Discrete Fourier Transform can also be used to multiply large numbers as well as perform a Fast Convolution. Some of the Transforms discussed here are invertible and so it makes sense to talk of the Inverse Transform.

The APL code in this paper is all in origin-0.

Hyperators

The sequence of objects: Arrays, Functions, and Operators appear in ascending order with the property that each later object consumes one or two of the earlier objects and produces an object in the sequence one order less.

For example, a Function takes Arrays as its **Argument(s)** and produces an Array; an Operator takes Arrays and/or Functions as its **Operand(s)** and produces a (derived) Function, which in turn takes Arrays as its Argument(s) and produces an Array.

Hyperators extend this sequence to the next level in that a Hyperator takes Arrays, Functions, and/or Operators as its **Hyperand(s)** and produces a (derived) Operator which in turn, etc.

That is, in the object hierarchy:

Order	Name	Options	Example
0	Array		
1	Function	0, 1, or 2	`∇Z←  f0`
		Arguments	`∇Z←  f1 R`
			`∇Z←L f2 R`
2	Operator	0, 1, or 2	`∇Z←  (LO op01)`
		Arguments	`∇Z←  (LO op11) R`
			`∇Z←L (LO op21) R`
		and	`∇Z←  (LO op02 RO)`
		1 or 2	`∇Z←  (LO op12 RO) R`
		Operands	`∇Z←L (LO op22 RO) R`
3	Hyperator	0, 1, or 2	`∇Z←  (LO (LH hy011))`
		Arguments	`∇Z←  (LO (LH hy111)) R`
			`∇Z←L (LO (LH hy211)) R`

		and
		
		1 or 2 Operands
			`∇Z←  (LO (LH hy021) RO)`
			`∇Z←  (LO (LH hy121) RO) R`
			`∇Z←L (LO (LH hy221) RO) R`

		and
		
		1 or 2 Hyperands
			`∇Z←  (LO (LH hy012 RH))`
			`∇Z←  (LO (LH hy112 RH)) R`
			`∇Z←L (LO (LH hy212 RH)) R`

			`∇Z←  (LO (LH hy022 RH) RO)`
			`∇Z←  (LO (LH hy122 RH) RO) R`
			`∇Z←L (LO (LH hy222 RH) RO) R`

Each Function, Operator, and Hyperator may have any of its lower order objects as its Arguments, Operands, or Hyperands, respectively. Missing from the above table due to lack of space is the option for both Operators and Hyperators of using a Jot (∘) for one or both Operands or Hyperands to indicate a missing element. Also missing are the usual function header enhancements such as Axis Operator, Optional Left Argument, Shy Result, Lists of Names in the Arguments (but not in Operands nor Hyperands), etc.

Anonymous Hyperators

Following the lead of Scholes[4], Anonymous Functions are extended to Hyperators by defining three new special symbols:

⍺⍺⍺	Left Hyperand
⍵⍵⍵	Right Hyperand
∇∇∇	Hyperator

In the case of Hyperators,

- ∇ references the Hyperator with both its Hyperand(s) and Operand(s) already bound (that is, ∇ references the entire derived function)

- ∇∇ references the Hyperator with only its Hyperand(s) bound (that is, ∇∇ references the Operator part of the Hyperator)
- ∇∇∇ references the Hyperator with neither its Hyperand(s) nor Operand(s) bound (that is, ∇∇∇ references the Hyperator itself)

In other words,

- for anonymous monadic hyperand monadic operand Hyperators,
```
    ∇ ←→ ⍺⍺∇∇     ←→ ⍺⍺ ⍺⍺⍺∇∇∇
```
- for anonymous monadic hyperand dyadic operand Hyperators,
```
    ∇ ←→ ⍺⍺∇∇⍵⍵ ←→ ⍺⍺ ⍺⍺⍺∇∇∇⍵⍵
```
- for anonymous dyadic hyperand monadic operand Hyperators,
```
    ∇ ←→ ⍺⍺∇∇     ←→ ⍺⍺ ⍺⍺⍺∇∇∇⍵⍵⍵
```
- for anonymous dyadic hyperand dyadic operand Hyperators,
```
    ∇ ←→ ⍺⍺∇∇⍵⍵ ←→ ⍺⍺ ⍺⍺⍺∇∇∇⍵⍵⍵ ⍵⍵
```

Transform Template

A **Transform Hyperator** is an instantiation into a single object of the mathematical concept of a **Transform**, examples[5,6] of which include Fourier[7], Fourier Sine[8], Fourier Cosine[8], Hartley[9], Laplace[10], Mellin[11], Stirling[12], Weierstrass[13], Binomial[14], Chebyshev[15], etc. Because we're dealing with discrete computer arithmetic, the Templates discussed in this paper are all in the category of **Discrete Transforms**[16], as opposed to continuous Integral Transforms.

For example, a Discrete Fourier Transform (DFT) of a vector of numbers may be written as:

```
    DFT ← {⍵+.×*(⍳⍴⍵)∘.×(⍳⍴⍵)×○0J¯2÷⍴⍵}                (1)
```

or more generally as a Dyadic Operator by splitting out the Inner Product functions (+ and ×) as Operands (⍺⍺ and ⍵⍵) because some Transforms use different Operands:

```
    DFT ← +{⍵ ⍺⍺.⍵⍵*(⍳⍴⍵)∘.×(⍳⍴⍵)×○0J¯2÷⍴⍵}×          (2)
```

or more generally by splitting out the Transform function

```
    hDFT ← {*(ιρω)∘.×(ιρω)×○0J¯2÷ρω}                    (3)
    DFT ← +{ω αα.ωω hDFT ω}×                              (4)
```

or more generally by writing the Transform function as the left Hyperand of a Monadic Hyperand Dyadic Operand Hyperator:

```
    DFT ← +hDFT{ω αα.ωω ααα ω}×                           (5)
```

Finally, with (⍑ – U+2351, Alt-’B’ on the keyboard) as its symbol[17] – this is the definition of a **Transform Hyperator**:

```
    ⍑ ↔ {ω αα.ωω ααα ω}   or equivalently as              (6)
      ↔ {αα.ωω∘ααα⍨ω}                                     (7)
```

Putting this all together,

```
    DFT ← +hDFT⍑×                                         (8)
```

That is, the template for a Transform Hyperator is

```
    {αα ααα⍑ωω ω}                                         (9)
```

where αα and ωω are the Inner Product Operands and ααα is the Transform Hyperator’s Hyperand, that is, the Transform function itself.

More Transforms

Other Transforms include the following, where we list only the Hyperands (h) to be used in the template `+h⍑×` :

```
    hFourierSine ← {1○(1+ιρω)∘.×○(1+ιρω)÷1+ρω}           (10)
    hFourierCosine ← {2○(0.5+ιρω)∘.×○(ιρω)÷ρω}           (11)
    hHartley ← {+⌿2 1∘.○(ιρω)∘.×(ιρω)×○2÷ρω}             (12)
    hLaplace ← {*(ιρω)∘.×-ιρω}                            (13)
    hMellin ← {(1+ιρω)∘.*ιρω}                             (14)
    hStirling ← {102‼¨(ιρω)∘.,ιρω}                        (15)
```

where `102‼M N` calculates Stirling Numbers of the 2nd kind.

```
    hWeierstrass ← {*¯0.25×((⍳⍴⍵)∘.-⍳⍴⍵)*2}         (16)
```
etc.

The following Transforms use a template of `-h⍑×` (in which case, the Inner Product reduction function is an Alternating Sum):

```
    hBinomial ← {(⍳⍴⍵)∘.!⍳⍴⍵}                       (17)
    hChebyshev ← hFourierCosine                     (18)
```

MORE TO BE WRITTEN IN THIS SECTION

Inverses

Note that the Inverse DFT can be expressed as a Hyperator, too:

```
    iDFT ← +{(*(⍳⍴⍵)∘.×(⍳⍴⍵)×○0J2÷⍴⍵)÷⍴⍵}⍑×          (19)
```

where the `DFT` and `iDFT` Hyperands differ only slightly:

```
    hDFT  ← { *(⍳⍴⍵)∘.×(⍳⍴⍵)×○0J¯2÷⍴⍵    }          (20)
    hiDFT ← {(*(⍳⍴⍵)∘.×(⍳⍴⍵)×○0J2 ÷⍴⍵)÷⍴⍵}          (21)
```

For example,

```
    DFT 1 2 3 4
10 ¯2J2 ¯2 ¯2J¯2
    iDFT DFT 1 2 3 4
1 2 3 4
```

MORE TO BE WRITTEN IN THIS SECTION

Applications

One interesting way in which the DFT and its inverse can be combined is to produce a Fast Convolution Algorithm[18]. Using the dyadic Convolution operator (⍡) built into NARS2000, the two concepts are related as described in this Wikipedia article[18]. The following code was adapted from Roger Hui's article[19]. The above Wikipedia article shows that Convolution "of two finite-length sequences is found by taking an FFT of each sequence, multiplying pointwise, and then performing an inverse FFT". Here we are using the notation of `DFT` where it is understood that an implementation of `DFT` and `iDFT` would translate the problem into FFT and iFFT.

The underlying identity from the Wikipedia article is

```
      a+⍡×b ←→ iDFT (DFT a2) × DFT b2
```

where `a2` and `b2` are extensions of `a` and `b` meant to be long enough to hold the number of digits in the Convolution of `a` and `b`.

For example,

```
      a←5 9 6 0 6 1 9 4 ⋄ b←8 5 2 5 4 1

      ⎕←c←a+⍡×b
40 97 103 73 125 109 122 115 67 63 57 25 4

      N←¯1+(⍴a)+⍴b
      (a2 b2)←N↑¨a b
      d←iDFT (DFT a2) × DFT b2
      ⌊0.5+9○d
40 97 103 73 125 109 122 115 67 63 57 25 4
```

In practice, `DFT` and `iDFT` return Complex floating point numbers some of whose Imaginary parts are small but non-zero. Because we know that the all items in the result should be not only Real numbers but also Integers, it's safe to ignore the Imaginary parts (`9○`) and round the Real parts (`⌊0.5+`) by prefacing the expression with `⌊0.5+9○` which produces the above result.

Taking this example one step farther, the result of the Convolution (and equivalently the inverse `DFT` of the product of two `DFTs`) can be expressed as a Multiple Precision Integer/Rational number in that the vectors `a`, `b`, and `c` represent the coefficients of a base `10` polynomial – that is, an Integer. The polynomial coefficients of the result in `c` may be greater than `9`, which means that each coefficient's tens digit and beyond must be carried into the coefficient to its left. This also explains why in the code below we need to Catenate a leading zero to `c` before performing the carry so as to the catch the overflow from the leftmost coefficient. Were the values in `c` much larger than two digits, we might need to Catenate more leading zeros to `c` in order to handle the overflow. The `carry` function is supplied by Hui in his article as

```
      carry←{1↓+⌿1 0⌽0,0 10⊤⍵}
```

and used on the result as

```
      carry⍣≡0,c
```

Note that the problem of carrying digits to the lefthand position can also be solved by recasting it as reducing the vector `0,c` using the following function

```
      carry2←{((⍺,0)+0 10⊤⊖⍴⍵),1↓⍵}
```

as an operand to reduction as in

```
      ⊃carry2/0,c
```

This line is functionally equivalent to `carry⍣≡0,c`, but much slower.

The following code demonstrates an example of using `DFT` to multiply two numbers in two ways – by Convolution and by its equivalent operation of `iDFT` of the product of two `DFTs`. Each number is represented as a vector of its coefficients.

The reason for this example becomes clear when you realize that `DFT` and `iDFT` would be implemented as a Fast Fourier Transform (FFT) and its inverse which are both significantly faster than the Discrete form. This technique is used especially in Multiple Precision libraries when multiplying very large Integers as one of several ways to make such multiplication practical.

```
      a←5 9 6 0 6 1 9 4 ⋄ b←8 5 2 5 4 1

      ⎕←a1←10x⊥a
59606194
      ⎕←b1←10x⊥b
852541
      a1×b1
50816724238954
      c←a+⍡×b
      10x⊥carry⍣≡0,c
50816724238954

      N←¯1+(⍴a)+⍴b
      (a2 b2)←N↑¨a b
      d←iDFT (DFT a2) × DFT b2
      10x⊥carry⍣≡0,⌊0.5+9○d
50816724238954
```

MORE TO BE WRITTEN IN THIS SECTION

System Function

As a temporary measure, the released version of NARS2000 provides a system function `⎕DFT` which implements the Discrete Fourier Transform using a Fast Fourier Transform. This function supports as input all Real and Complex datatypes.

This implementation employs three different sources of algorithms for FFT and its inverse:

- Gnu Scientific Library (GSL) for Fixed Precision arguments,
- MPFFT for MPIR/MPFR Multiple Precision arguments, and
- ARB for Ball Arithmetic arguments,

so there may be some slight differences when comparing the results across datatypes.

A monadic call to `⎕DFT` calculates a Discrete Fourier Transform (using the usual very fast FFT), as does a dyadic call with a left argument of `1`. A dyadic call with a left argument of `¯1` calculates the inverse `DFT` (this time using the usual very fast Inverse FFT).

Extending the arguments to a length which is a power of two (as required by the FFT algorithm for maximum efficiency) is handled within the system function.

This system function is temporary only – when the idea of Hyperators comes to fruition, the system function will be removed in favor of calling the corresponding Transform Hyperator.

MORE TO BE WRITTEN IN THIS SECTION

Timings

FWIW, using the form

```
      ¯1 ⎕DFT(⎕DFT a2)×⎕DFT b2
```

on million-digit numbers `a2` and `b2` (represented as million-element vectors of 64-bit integers `0` through `9`), takes less than `7` seconds to calculate. Using the equivalent (but slower) method of Convolution (a+⍡×b) takes so much time as to exhaust my patience. At the other end of the timing extreme, multiplication of the same numbers represented as million-digit Multiple Precision Integers (a1×b1), takes less than 0.1 seconds.

```
      (a1 b1)←?2⍴10*1E6x
      ⎕T-(a1×b1)⊢⊢⎕T
0.06809538643574342
```

MORE TO BE WRITTEN IN THIS SECTION

Implementation

The above listed Hyperands along with their corresponding Operands are entirely executable, however that might not yield the highest performance. Instead the Hyperand/Operand combination can serve as a recognizable template which then triggers execution under the hood of a higher performance algorithm, such as the aptly named (and significantly faster) Fast Fourier (and Inverse) Transforms in place of the Discrete Fourier Transform and its Inverse.

For example, the implementation of the Transform Hyperator (⍑) checks its two Operands and one Hyperand – if they are identically +hDFT⍑×, or +hiDFT⍑× then the FFT or iFFT code is executed.

MORE TO BE WRITTEN IN THIS SECTION

Acknowledgments

No paper is written in isolation, and this paper is no exception. I'd like to thank **YOUR NAME GOES HERE** for their helpful advice, suggestions, and examples.

Online Version

This paper is an ongoing effort and can be out-of-date the next day. You may find the most recent version at http://sudleyplace.com/APL/ – look for the title of this paper there.

Executable Version

All of the above APL code may be executed in NARS2000, an experimental APL interpreter available for free as Open Source software. Also, a workspace is available which contains the above Hyperands and other code:

http://nars2000.org/download/workspaces/Hyperators.ws.nars

The latest Gamma version of NARS2000 implements Hyperators in the form of User-Defined[2] and Anonymous[3] Functions as well as The Transform Hyperator (⍑). It may be downloaded from http://www.nars2000.org/download/binaries/gamma/ in either 32- or 64-bit versions. This software runs natively under Microsoft Windows Win7 or later as well as any Linux or Mac OS version which supports Wine (32-bit only) which acts as a translation layer.

References

1. Hyperators, NARS2000 Wiki, http://wiki.nars2000.org/index.php/Hyperators

2. User-defined Functions, Operators, and Hyperators, NARS2000 Wiki, http://wiki.nars2000.org/index.php/User-Defined_Functions/Operators/Hyperators
3. Anonymous Functions, Operators, and Hyperators, NARS2000 Wiki, http://wiki.nars2000.org/index.php/Anonymous_Functions/Operators/Hyperators
4. “The San Quirico Moot”, 4-10 June 2007, “Hyper-operators”, John Scholes, http://archive.vector.org.uk/art10011760
5. Integral Transforms, Wikipedia, https://en.wikipedia.org/wiki/Integral_transform#Table_of_transforms
6. Integral Transforms, Wikipedia, https://en.wikipedia.org/wiki/List_of_transforms#Integral_transforms
7. Discrete Fourier Transform, Wikipedia, https://en.wikipedia.org/wiki/Discrete_Fourier_transform
8. Sine and Cosine Transforms, Wikipedia, https://en.wikipedia.org/wiki/Sine_and_cosine_transforms
9. Hartley Transform, Wikipedia, https://en.wikipedia.org/wiki/Hartley_transform
10. Laplace Transform, Wikipedia, https://en.wikipedia.org/wiki/Laplace_transform
11. Mellin Transform, Wikipedia, https://en.wikipedia.org/wiki/Mellin_transform
12. Stirling Transform, Wikipedia, https://en.wikipedia.org/wiki/Stirling_transform
13. Weierstrass Transform, Wikipedia, https://en.wikipedia.org/wiki/Weierstrass_transform
14. Binomial Transform, Wikipedia, https://en.wikipedia.org/wiki/Binomial_transform

15. Discrete Chebyshev Transform, Wikipedia, https://en.wikipedia.org/wiki/Discrete_Chebyshev_transform

16. Discrete Transforms, Wikipedia, https://en.wikipedia.org/wiki/List_of_transforms#Discrete_transforms

17. Note that the Unicode name for this symbol is UpTackOverbar (U+2351), not DownTackOverbar. For consistency with other related names in NARS2000, I will refer to it as DownTackOverbar.

18. Fast Convolution Algorithms, Wikipedia, https://en.wikipedia.org/wiki/Convolution#Fast_convolution_algorithms

19. Hui, Roger, Dyalog dfns workspace on “Fast multi-digit product using FFT”, http://dfns.dyalog.com/n_xtimes.htm