PDL

Basics of
Indexing and Threading

Outline

- Motivation
- Indexing
  - Dimension manipulation
  - Slicing
  - Parent and child relation
- Threading
  - Function's signature
  - The core and extra dimensions

References:

1. http://pdl.sourceforge.net/PDLdocs/Indexing.html
2. http://www.johnlapeyre.com/pdl/pdldoc/newbook/node5.html

Motivation

Optimized manipulation of multi-dimensional data structures.

This is achieved by automated looping over dimensions (called threading).

Indexing
Dimension Manipulation

Indexing allows a very flexible access to the data of a piddle.

First we need to know how to track and manipulate dimensions.

```
perldl> p $a = sequence(5,2);
 [0 1 2 3 4]
 [5 6 7 8 9]
perldl> p $a->dims;
5 2
perldl> p $a->ndims;
2
perldl> p $a->dim(0);
5
perldl> p $a->nelem;
10
```

Note – first columns then rows

dimension sizes

number of dimensions

size of the 0^{th} dimensions

number of elements

Indexing
Dimension Manipulation

Now let's do some shuffling...

```
perldl> p $a;
 [0 1 2 3 4]
 [5 6 7 8 9]
perldl> p $a->xchg(0,1);
 [0 5]
 [1 6]
 [2 7]
 [3 8]
 [4 9]
```

exchange the 0th and 1st dimensions

On a larger piddle:

```
perldl> $m = sequence(3,2,1,5,4);
perldl> p $m->dims;
3 2 1 5 4
perldl> p $m->xchg(0,2)->dims;
1 2 3 5 4
perldl> p $m->mv(1,3)->dims;
3 1 5 2 4
```

move the 1st dimension to be the 3rd dimension

Indexing
Dimension Manipulation

Adding dimensions:

```
perldl> p $x = sequence(3);
[0 1 2]
perldl> p $x->dims;
3
```

but this can also be represented as a (1,3) matrix:

```
perldl> p $x->dummy(0);
 [0]
 [1]
 [2]
perldl> p $x->dummy(0)->dims;
1 3
```

add a "dummy" 0th dimension of size 1 (default size)

and in PDL you can also do this:

```
perldl> p $y = $x->dummy(0,3);
 [0 0 0]
 [1 1 1]
 [2 2 2]
```

add a "dummy" 0th dimension of size 3

Indexing
Dimension Manipulation

Removing dimensions:

```
perldl> p $y;
 [0 0 0]
 [1 1 1]
 [2 2 2]

perldl> p $y->clump(2);
[0 0 0 1 1 1 2 2 2]
```

clump together first 2 dimensions

```
perldl> p $x = sequence(3)->dummy(1);
[
 [0 1 2]
]
perldl> p $x->squeeze;
[0 1 2]
```

Note: in other examples I erased the outer rectangular brackets

eliminate all dimensions of size 1
can also be done by `$x(;-)`

```
perldl> $x = sequence(2,2,2);
perldl> p $x->flat
[0 1 2 3 4 5 6 7]
```

flatten a piddle to a 1D piddle.
can also be done by `$x(;_)`

Indexing
Dimension Manipulation

Other dimension manipulation functions:

reorder – reorders the dimensions of a piddle.

splitdim – splits a dimension (the opposite of clump).

reshape – change the dimension of a piddle (note: physical (parent) piddles are changed inplace)

cat, glue, append...

Indexing
Slicing

The slice function enables the extraction of rectangular slices of piddles. PDL::NiceSlice enables a concise syntax (loaded automatically in perldl).

```
perldl> p $x = sequence(5,5);
 [ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]

perldl> p $x(:,0:1);
 [0 1 2 3 4]
 [5 6 7 8 9]
```

Extract the even elements along the 1st dimension:

```
perldl> p $x(,0:-1:2);
 [ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
```

Indexing
Slicing

Slice and reverse:

```
perldl> p $x(,3:1)
 [15 16 17 18 19]
 [10 11 12 13 14]
 [ 5  6  7  8  9]
```

Reminder: `$x` equals to

```
[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 11 12 13 14]
[15 16 17 18 19]
[20 21 22 23 24]
```

To extract the diagonal you can do:

```
perldl> p $x(0:-1:6;_);
 [0 6 12 18 24]
```

or just use the diagonal function...

and you can also extract elements without any periodicity:

```
perldl> $idx = pdl(4,0,1);
perldl> p $x($idx,$idx);
 [24 20 21]
 [ 4  0  1]
 [ 9  5  6]
```

Indexing
Slicing

Slicing using conditions:

```
perldl> p $x($x>17;?);
[18 19 20 21 22 23 24]
```

this can also be obtained by:

```
perldl> p $x->where($x>17);
```

Using multiple conditions:

```
perldl> p $x($x>17 & $x<20;?);
[18 19]
perldl> p $x($x>17 | $x<5;?);
[0 1 2 3 4 18 19 20 21 22 23 24]
```

Reminder: `$x` equals to

```
[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 11 12 13 14]
[15 16 17 18 19]
[20 21 22 23 24]
```

Indexing
Parent-Child Relation

```
perldl> p $x = sequence(3,3);
 [0 1 2]
 [3 4 5]
 [6 7 8]
```

Here defining a new piddle.
This is called now the “parent”.

```
perldl> p $line = $x(:,2;-);
[6 7 8]
```

Here defining a new piddle to be a slice of the “parent”. This is called a “child”.
(note that without the “-” we had a 2D piddle)

```
perldl> p $line++
[7 8 9]
```

Making some changes to the child...

```
perldl> p $x
 [0 1 2]
 [3 4 5]
 [7 8 9]
```

changes also the parent.

For assignments use .=

The dataflow between the child and the parent is bidirectional → enables the simultaneous representation of the same data in several different ways.

Indexing
Parent-Child Relation

A child does not consume extra memory (as with references). Therefore it is called a “virtual piddle”.

The dataflow between a parent and child can be broken in two ways:

1. sever – severs any links of a piddle to its parents.
2. copy – creates a physical copy of a piddle.

In most cases they operate similarly, but they act differently on parent piddles: sever will do nothing and copy will create a new physical copy.

Indexing
Parent-Child Relation

An exampe for sever:

```
perldl> $a = zeroes(5);
perldl> $b = $a(1:3);
perldl> $b++;
perldl> p $a;
[0 1 1 1 0]
perldl> $b->sever;
perldl> p $b++;
[2 2 2]
perldl> p $a;
[0 1 1 1 0]
```

Shorthand format:
use
`$b = $a(1:3;|);`
instead of
`$b = $a(1:3)->sever;`

Threading

- Threading in PDL means an implicit looping facility.
- It allows fast processing of large amounts of data.
- It is not (directly) related to threading in the computer science sense.

Threading

A simple example:

The function maximum is defined to find the maximal element along a 1D piddle. Threading allows it to be run on piddles of any dimension, without any syntactical effort:

```
perldl> p $a = sequence(3);
[0 1 2]
perldl> p $a->maximum
2
perldl> p $a = sequence(3,3);
 [0 1 2]
 [3 4 5]
 [6 7 8]
perldl> p $a->maximum
[2 5 8]
```

so how does this work → → →

Threading

We need to understand:

1. The elementary operation of a function (signatures).

2. How threading treats extra dimensions.

3. How to manipulate the default threading operation (dimension manipulation).

Threading
Signatures

The definition of a function's input and output dimensions appears in the function's signature:

```
perldl> sig maximum
  Signature: maximum(a(n); [o]c())
```

This information can also be found using "? maximum"

- `a` is an input piddle, `c` is an output piddle (the names don't matter).
- `(n)` stands for the dimension of the input, which can be any 1D piddle.
- `[o]` stands for output.
- `()` means zero-dimension (a scalar).

This signature tells us that "maximum" expects a 1D piddle as input and returns a zero-dimensional piddle (a scalar) as output.

Threading
Signatures

Let's look at another function – inner:

```
perldl> sig inner
  Signature: inner(a(n); b(n); [o]c())
```

This signature tells us that inner expects two 1D piddles of the same dimension size and returns a scalar.

Threading
The Extra Dimensions

What happens if we provide a function with piddles that have more dimensions than defined in the function's signature?

In this case threading takes care of the extra dimensions.

Definitions:

1. *Core dimensions* – the dimensions which are required by the signature. By default they are the first dimensions of the piddle.
2. *Loop (or extra) dimensions* – all the other dimensions over which the function is being looped ("threaded") over.

Threading

The Extra Dimensions

Case 1: an example for the core and loop dimensions – 1 input argument

```
perldl> sig maximum
  Signature: maximum(a(n); [o]c())
perldl> p $a = sequence(4,3);
 [ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
perldl> p $a->maximum;
[3 7 11]
```

Here the core dimension is the 0th dimension (columns) of size 4.

maximum is threaded over these slices:

```
perldl> p $a(:,0);
 [0 1 2 3]
perldl> p $a(:,1);
 [4 5 6 7]
perldl> p $a(:,2);
 [8 9 10 11]
```

→ the 1st dimension is a loop dimension

Threading

The Extra Dimensions

When the elementary output is a scalar, the number and size of the output dimensions are as that of the extra dimensions.

In the last example: 1D piddle of size 3.

Reminder:

```
perldl> sig maximum
  Signature: maximum(a(n); [o]c())
perldl> p $a = sequence(4,3);
 [ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
perldl> p $a->maximum;
[3 7 11]
```

Threading

The Extra Dimensions

Case 2: an example with more than one input argument

```
perldl> sig inner
  Signature: inner(a(n); b(n); [o]c())

perldl> p $a = sequence(3,2);
 [0 1 2]
 [3 4 5]

perldl> p $b = ones(3);
[1 1 1]

perldl> p inner($a,$b);
[3 12]
```

The 0^{th} dimension of $a and of $b match as required by inner.

But - $a has 1 extra dimension of size 2 while $b doesn't.

Threading
The Extra Dimensions

Threading takes care of this missing dimension automatically, but you can think of this as if a dummy 1st dimension of size 2 was added to $b:

```
perldl> p $b->dummy(1,2)
 [1 1 1]
 [1 1 1]
```

and now all dimensions match.

Reminder:

```
perldl> $a = sequence(3,2);
 [0 1 2]
 [3 4 5]
perldl> p $b = ones(3);
[1 1 1]
perldl> p inner($a,$b);
[3 12]
```

Threading

Manipulating Dimensions

Case 3: an example for a case where the core dimensions of the input piddles don't fit the signature's

```
perldl> sig inner
  Signature: inner(a(n); b(n); [o]c())
perldl> p $a = sequence(2,3);
 [0 1]
 [2 3]
 [4 5]
perldl> p $b = ones(3);
[1 1 1]
perldl> p inner($a,$b);
Error in inner:Wrong dims
```

The first dimensions don't match as required by the signature - we get an error.

Threading
Manipulating Dimensions

Reminder:

```
$a =
 [0 1]
 [2 3]
 [4 5]
$b =
[1 1 1]
```

There are two ways to resolve this:

1. Add a dummy 0^{th} dimension to $b:

```
perldl> p $b->dummy(0,2);
 [1 1]
 [1 1]
 [1 1]
```

and now the dimension of $b and $a match.

2. Exchange the dimensions of $a to get a piddle of dimension size (3,2) (which here is the same as using transpose):

```
perldl> p $a->xchg(0,1);
 [0 2 4]
 [1 3 5]
```

and we're back to case 2.

Threading

Case 3: an example with multiple core and extra dimensions (taken from PDL::Indexing page – ref 1)

Let's assume we have a function with the following signature:

func((m, n) , (m, n, k), (m), [o](m, k))

This function expects three piddles as input with the above specified dimensions and returns an output piddle with the corresponding dimensions.

Now, what happens if we supply this function with piddles of the following dimensions:

a(5, 3, 10, 11) b(5, 3, 2, 10, 1, 12) c(5, 1, 11, 12)

The sizes of the core dimensions are: m = 5, n = 3, k = 2
and they match as required.

Threading

What are the loop dimensions (LD)?

signature: func((m, n) , (m, n, k), (m), [o](m, k))

a(5, 3, 10, 11) b(5, 3, 2, 10, 1, 12) c(5, 1, 11, 12)

- According to the dimensions of a: first LD size is 10, second is 11.
- Checking if b LDs match: first LD size is 10 – match, second is 1 – this will automatically be extended to 11, and there is a third LD of size 12.
- Checking the dimensions of c: first LD size is 1 – will automatically be extended to 10, second is 11 – match, third is 12 - match.

Threading

To summarize:

signature: func((m, n) , (m, n, k), (m), [o](m, k))

a(5, 3, 10, 11) b(5, 3, 2, 10, 1, 12) c(5, 1, 11, 12)

The core dimensions are: m = 5, n = 3, k = 2

The loop dimensions are: 10, 11, 12

The output dimensions will be: 5, 2, 10, 11, 12

The End

For further reading see the references:

1. http://pdl.sourceforge.net/PDLdocs/Indexing.html
2. http://www.johnlapeyre.com/pdl/pdldoc/newbook/node5.html

Exercise

1. Find the maximal element of each column of a 2D matrix.

2. Extract the odd elements along the columns of a 2D matrix.

And now for a real challenge:

Calculate the tensor product of two matrices.