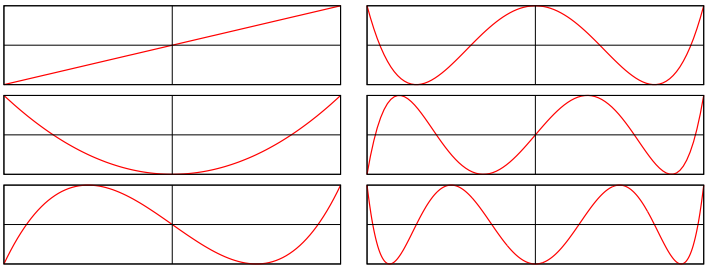
An embedded DSL for Chebyshev interpolation

Björn von Sydow

DSLs of Math, February 12, 2016.



What we want to do

The problem

We want to develop a problem-solving environment for calculus, where common computational tasks can be easily expressed and efficiently computed.
In a first step, we consider real-valued functions on the interval $-1 \leqslant x \leqslant 1$.

What computational tasks?

We want to be able to compute at least

- derivatives,
- primitive functions and integrals,
- roots of functions,
- maxima/minima of functions.

Later: solve differential equations.

The language we seek

We want something like

```
type RealFun = Double -> Double

diff     :: RealFun -> RealFun
prim     :: RealFun -> RealFun
roots    :: RealFun -> [Double]
maxima   :: RealFun -> [Double]
integral :: RealFun -> Double
```

Preliminaries 1: Representing vectors in programming languages

The problem

We will need mathematical vectors, such as $(\ 1 \quad 1 \quad 2 \quad 0.5\ ) \in \mathbb{R}^4$. Which type should we use?

It must work well in $\mathbb{R}^n$, also for large n.

Java arrays

```
double[] v = {1.0,1.0,2.0,0.5};
...
static double[] scalProd(double[] a,
                         double[] b){
  double sum = 0.0;
  for (int i=0; i<a.length; i++)
     sum = sum + a[i]*b[i];
  return sum;}
```

Haskell lists

```
v = [1.0,1.0,2.0,0.5]
...
scalProd::[Double]->[Double]->[Double]
scalProd a b = sum (zipWith (*) a b)
```

Which of these two types is "better"?

Key differences: Java arrays

1. have constant-time access.
2. have less memory overhead.
3. are mutable.

The two first of these are obviously advantages; what about the third?

Preliminaries 2: Vectors in Haskell

The *vector* package

Provides several types of efficient vectors. We will use type `Vector` from `Data.Vector.Storable`.

These are immutable, constant-time access vectors with small overhead and powerful optimizations.

The language of `Vectors`

Many familiar functions on lists have counterparts for `Vector`, with the same names.
Examples: `map`, `take`, `drop`, `sum`, `foldl`, `zipWith`, `elem`, *etc.*

Example interaction

```
> import qualified Data.Vector.Storable as V
> let v = V.enumFromTo 1 5
> v
[1.0,2.0,3.0,4.0,5.0]
> v V.! 3
4.0
> V.sum v
15.0
> V.map sqrt v
[1.0,1.4142135623730951,1.7320508075688772,
2.0,2.23606797749979]
> V.head v
1.0
```

Note `qualified` import to avoid name clashes!

Preliminaries 3: What is the type of +?

Interaction

```
> 3+4
7
> 3+pi
6.141592653589793
> v+v
...type error...
> :t (+)
Num a => a -> a -> a
```

New interaction

```
> v+v
[2.0,4.0,6.0,8.0,10.0]
> v+1
[2.0,3.0,4.0,5.0,6.0]
```

Check that you understand the last computation!

Vectors as "numbers"

```
instance (V.Storable a,
          Num a) => Num (V.Vector a) where
   (+)           = zW (+)
   (-)           = zW (-)
   (*)           = zW (*)
   abs           = V.map abs
   negate        = V.map negate
   signum        = V.map signum
   fromInteger n = V.singleton (fromInteger n)

 zW f as bs
  |la == lb  = V.zipWith f as bs
  |la == 1   = V.map (f (V.head as)) bs
  |lb == 1   = V.map (flip f (V.head bs)) as
  |otherwise = error "incompatible lengths"
  where la = V.length as; lb = V.length bs
```

Preliminaries 4: More type classes

More examples

```
> 3*v
[3.0,6.0,9.0,12.0,15.0]

> let scalProd a b = V.sum(a*b)
```

What about adding functions?

```
> map (sin+cos) [1..5]
... type error ....
```

Two functions cannot be added. So, recall from Lecture 6:

```
instance Num a => Num (b -> a) where
   f + g = \x -> f x + g x
   ...
   fromInteger = const . fromInteger
```

What is the type of `sqrt`?

```
>:t sqrt
sqrt :: Floating a => a -> a
```

We define another instance:

```
instance (V.Storable a, Floating a) =>
         Floating (V.Vector a) where
   pi    = V.singleton pi
   sqrt  = V.map sqrt
   ...
```

And now

```
> sqrt v
[1.0,1.4142135623730951,1.732050807568877,
2.0,2.23606797749979]
```

Aside: DSL's and type classes

Example: `Num`

The type class `Num` defines a language of (badly named) numerical expressions, such as

```
fromInteger 3 + abs (fromInteger (-4)).
```

These have type `Num a => a`.

By defining instances, we can give semantics to this language at different types.

How does this relate to shallow and deep embeddings?

Possible answer

- The type class mechanism is analogous to a deep embedding in the sense that no semantics is implied by class definition.
- Each instance gives (typically) a shallow embedding into the corresponding type.
- The choice of instance is guided by type, not by name of a run function.

Back to the main problem

What do we know how to do?

- `diff` can be done by automatic differentiation (Lecture 6, with continuation next week).
- For smooth functions we have

```
maxima f = V.filter
               (\x -> f'' x < 0)
               (roots f')
   where f'  = diff f
         f'' = diff f'
```

(This is not quite true; what is missing?)

The others are not obvious.

Polynomial approximation

We try the following approach:

1. Find a method to compute, for given f, a sequence of polynomials $P_0, P_1, P_2, \ldots$, that converges (uniformly) to f as $n \to \infty$.
2. Introduce a type `Poly` of polynomials and
   - define `diff`, `prim`, `roots` etc for `Poly` instead of `RealFun` (hopefully easier?!).
   - use P_n `::Poly`, for some suitable n, instead of f.

How do you find good polynomial approximants?

Idea 1: Taylor expansion

Use a Taylor polynomial around 0.
For e.g. $f(x) = \sin(x)$, we get

$$P_1(x) = x$$
$$P_3(x) = x - x^3/6$$
$$P_5(x) = x - x^3/6 + x^5/120$$
$$\ldots$$

Problems

- Requires high order derivatives for good approximation.
- Uses only info at $x = 0$; need not converge in all [-1,1].

Idea 2: Interpolating polynomial

Given $n+1$ points

$$x_0 < x_1 < \ldots < x_n,$$

we can find interpolating P_n such that $P_n(x_k) = f(x_k)$ for all k.

For example, take $x_k = 2k/n - 1$.

Problem

Will not converge throughout [-1,1] for most functions f.

Folklore: Polynomial interpolation is a bad idea.

Chebyshev polynomials

Definition

The Chebyshev polynomial of degree n is

$$T_n(x) = \cos(n \arccos x).$$

Are they polynomials at all?

Recall (from first-year calculus)

$$\cos(\alpha + \beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta,$$
$$\cos(\alpha - \beta) = \cos\alpha\cos\beta + \sin\alpha\sin\beta.$$

Add them and put $\alpha = n \arccos x$ and $\beta = \arccos x$ to get

$$T_{n+1}(x) + T_{n-1}(x) = 2xT_n(x)$$

Recursion scheme

$$T_0(x) = 1,$$
$$T_1(x) = x,$$
$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x), \quad n \geqslant 1.$$

Properties

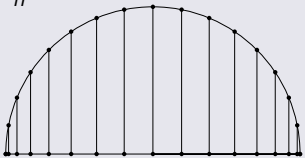
- T_n is a polynomial of degree n.
- $-1 \leqslant T_n(x) \leqslant 1$ for $-1 \leqslant x \leqslant 1$.
- $|T_n(x)| = 1$ at $x_k = \cos\frac{k\pi}{n}$ for $k = 0, 1, \ldots n$, with alternating sign.
- T_n has n zeros in $-1 \leqslant x \leqslant 1$.

Chebyshev interpolation

Key insight 1

Use interpolation, but choose $\{x_k\}$ as the Chebyshev points:

$x_k = \cos \frac{k\pi}{n}$, $k = 0, \ldots n$.



Convergence

- If $f \in C^k([-1, 1])$, then $\|f - P_n\|_\infty = O(n^{-k})$.
- If f is analytic, then $\|f - P_n\|_\infty = O(\rho^{-n})$ for some $\rho > 1$.

Key insight 2

Represent P_n in the Chebyshev basis, i.e. as $P_n = \sum_{k=0}^{n} c_k T_k$.

We need to compute $\{c_k\}$ so that $P_n(x_k) = f(x_k)$ for all k.

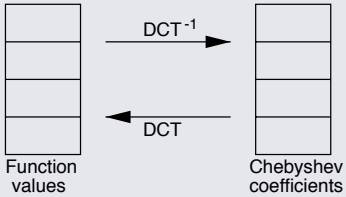
Computation

- $[c_0, c_1, \ldots c_n]$ is the inverse DCT of $[f(x_0), f(x_1), \ldots f(x_n)]$.
- There are simple, linear, stable recursions for computing
  - $P_n(x)$ (Clenshaw's algorithm).
  - $P_n'(x)$.
  - $\int^x P_n(t)dt$.

More on algorithms, 1

Discrete cosine transform

A variant of the Discrete Fourier transform.



Can be computed by the Fast Fourier Transform (FFT), with computing time $O(n \log n)$.

We use a fast C library, FFTW, accessed through Haskell wrappers.

Clenshaw's algorithm

Problem: Given $[c_0, \ldots c_n]$ and x, compute $\sum_{k=0}^{n} c_k T_k(x)$.

- Use $T_n = 2xT_{n-1} - T_{n-2}$; we get

$$\sum_{k=0}^{n} c_k T_k(x) = \sum_{k=0}^{n-1} c'_k T_k(x)$$

where

$$\begin{aligned} c'_{n-1} &= c_{n-1} + 2xc_n \\ c'_{n-2} &= c_{n-2} - c_n \\ c'_k &= c_k \text{ for } k < n-2. \end{aligned}$$

- Repeat until degree is 1.
- Use $T_1(x) = x$, $T_0(x) = 1$.

More on algorithms, 2

Root-finding

Given $[c_0, c1, \ldots c_n]$, we can define a $n \times n$ matrix C_n, such that the roots of $\sum_{k=0}^{n} c_k T_k(x)$ are the eigenvalues of C_n.

Good algorithms for eigenvalue computation exist. To use these, we need to deal efficiently also with matrices in Haskell.

We use the fast C library LAPACK, accessed through Haskell wrappers.

Integrals

Using $T_n(x) = \cos(n \arccos x)$, one gets

$$\int_{-1}^{1} T_k(x)dx = \frac{2}{1-k^2} \text{ for even } k$$

(the integral is 0 for odd k). Thus

$$\int_{-1}^{1} \sum_{k=0}^{n} c_k T_k(x)dx = \sum_{k=0, k \text{ even}}^{n} \frac{2c_k}{1-k^2}.$$

Note that this equality is exact!

Chebfuns

The type `ChebFun`

```
data ChebFun =
     BaseFun (Vector Double)
   | ... other constructors
```

`BaseFun cs`, where $\texttt{cs} = [c_0, c_1, \ldots c_n]$, represents the polynomial $\sum_{k=0}^{n} c_k T_k$,

Other constructors

- For intervals other than $[-1, 1]$.
- For non-smooth functions (several pieces glued together).

Constructing a `ChebFun`

```
chebfun :: (Vector Double -> Vector Double)
           -> ChebFun
```

`chebfun f` attempts to construct a polynomial interpolant that approximates f to (almost) machine precision.
Successively higher degrees are tried until error small.

`ChebFun` is an instance of the numeric type classes

So, we can do e.g.

```
let x = chebfun id
let c = sin (exp x) + sqrt x
```

An alternative view of what we have done

```
class Hatlab a where
  fromFun  :: (Vector Double -> Vector Double) -> a
  eval     :: a -> Vector Double -> Vector Double
  var      :: (Double, Double) -> a

  diff     :: a -> a
  prim     :: a -> a
  maxC     :: a -> a -> a
  minC     :: a -> a -> a

  roots    :: a -> Vector Double
  maxima   :: a -> Vector Double
  integral :: a -> Double
```

The `Hatlab` class specifies the computational tasks we consider.

The `ChebFun` instance is a fast, high-precision, shallow embedding of this language.

Differential equations: introducing spectral methods

Example: Airy equation of order 1

$y'' - xy = 0$, with boundary conditions $y(-20) = 1, y(20) = 0$.

Differential operators

```
class Dif a where
  dif :: Int -> a -> a

data Chebop = Chebop {
      dom :: (Double,Double),
      op  :: ... type omitted ...
      bcs :: [BoundaryCondition]
    }
```

Note: RHS not part of operator.

Boundary conditions

```
data BC = Dir Double | Neu Double
type BoundaryCondition = Either BC BC
```

Dirichlet condition: value of y prescribed.
Neumann condition: value of y' prescribed.

The Airy example

```
airy = Chebop (-20,20)
          (\x y -> dif 2 y - x*y)
       [Left (Dir 1), Right (Dir 0)]
```

Differentiation matrices

Differentiation on a grid

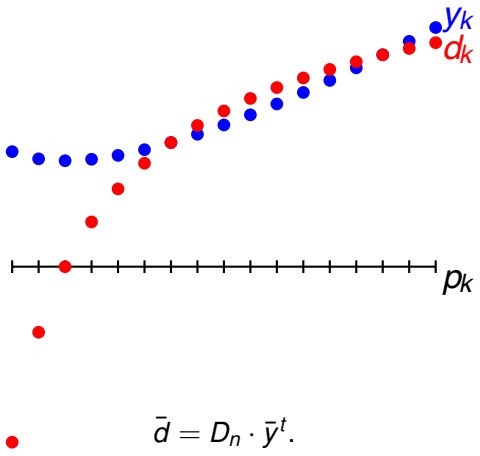
Fix $n+1$ points $\{p_k\}_{k=0}^n$ on the real line.

We define an operator $\mathcal{D}_n$ on $\mathbb{R}^{n+1}$:

Given $\bar{y} = (y_0\ y_1\ \ldots\ y_n) \in \mathbb{R}^{n+1}$,

1. Find the polynomial P of degree at most n, such that $P(p_k) = y_k$ for all k.
2. Let P' be the derivative of P.
3. Then $\mathcal{D}_n(\bar{y}) = \bar{d}$, where $\bar{d} = (P'(p_0)\ P'(p_1)\ \ldots\ P'(p_n))$.

$\mathcal{D}_n$ is a linear operator, so there is a $(n+1) \times (n+1)$ matrix D_n such that $\bar{d} = D_n \cdot \bar{y}^t$.



Differentiation on the Chebyshev grid

The Chebyshev grid

For best convergence, choose $\{p_k\}_{k=0}^n$ as the Chebyshev points of order n. For this grid, explicit formulas for D_n are known.

Example: D_4 is

$$\begin{pmatrix} 5.500 & -6.828 & 2.000 & -1.172 & 0.500 \\ 1.707 & -0.707 & -1.414 & 0.707 & -0.293 \\ -0.500 & 1.414 & 0.000 & -1.414 & 0.500 \\ 0.293 & -0.707 & 1.414 & 0.707 & -1.707 \\ -0.500 & 1.172 & -2.000 & 6.828 & -5.500 \end{pmatrix}$$

Differential operators as matrices

Example: The Airy operator on the Chebyshev grid is

$$\left(D_n^2 - \begin{pmatrix} x_0 & & & \\ & x_1 & & \\ & & \ddots & \\ & & & x_n \end{pmatrix} \right) \cdot \bar{y}^t$$

where $\bar{x} = (x_0\ x_1 \ldots x_n)$ and $\bar{y}$ are the values of $\lambda x \to x$ and y on the grid.

Solving differential equations

Algorithm

To solve $\mathcal{L}y = f$, where $\mathcal{L}$ is a differential operator:

1. Choose n and find the matrix L_n representing $\mathcal{L}$ on the Chebyshev grid of order n
2. Compute $\bar{f}$, the values of the RHS on the grid.
3. Modify L_n to handle boundary conditions (omitted).
4. Solve the linear system $L_n \cdot \bar{y}^t = \bar{f}^t$.
5. Check convergence; if not OK, double n and repeat.

Convergence

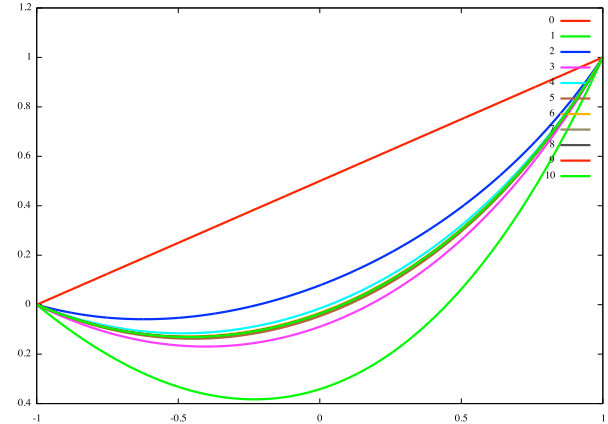
The spectral method on the Chebyshev grid converges rapidly for smooth problems. Can often get higher precision results than with other methods.

Further examples for demo.

- Bessel's equation, order 1.
  $x^2y'' + xy' + (x^2 - 1)y = 0$,
  $y(0) = 0, y(40) = 1$.
- $y'' + 50(1 + \sin x)y = 1$,
  $y(-20) = y(20) = 0$.

A non-linear example: $u'' = \exp(u), u(-1) = 0, u(1) = 1$

```
d = chebop (-1,1) (\x u -> dif 2 u) [Left (Dir 0), Right (Dir 1)]
us = iterate (\u -> d <\> exp u) ((var (-1,1) + 1)/2)
plot [Cheb (us!!k) (show k) | k <- [0..10]]
```



Iterated solution

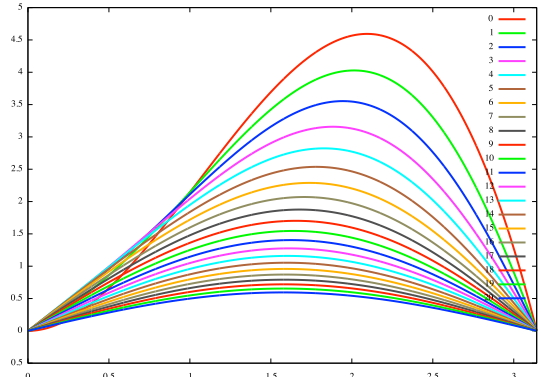
Starting from a linear function satisfying boundary conditions, we iterate towards the fixpoint.

```
> let b = us!!30
> norm_2 (dif 2 b - exp b)
2.332516486097704e-11
```

Stopping condition easily added.

The heat equation: $u_t = u_{xx}, \quad u(x,0) = f(x), u(0,t) = u(\pi,t) = 0.$

```
d tau = chebop (0,pi) (\x u -> con tau * dif 2 u - u) [Left (Dir 0), Right
f x = x^2 * (pi - x)
us = iterate (\u -> d 0.1 <\> (-u)) (f (var (0,pi)))
plot [Cheb (us!!k) (show k) | k <- [0..20]]
```



Iterated solution

Time-stepping using “Euler backwards”:

$$\frac{u^{n+1} - u^n}{\tau} = u_{xx}^{n+1}$$

Solve a linear ODE at each time step.
$\tau = 0.1$ used in example run.

Credits and references

Credits

The beautiful mathematical theory is classical and has many contributors.

The usefulness for scientific computing has been strongly advocated by Nick Trefethen and coworkers, mainly through the Matlab package Chebfun and associated papers.

We have only adapted their Chebfun ideas to Haskell.

References

- Nick Trefethen: Six myths about polynomial interpolation and quadrature. Summer Lecture at the Royal Society 2011. Easily found by googling. Compulsory reading!
- Nick Trefethen: Approximation Theory and Approximation Practice. SIAM Press 2013.