Disciplined Convex Programming and Symbolic Subdifferentiation in Haskell

Chris Copeland    Mickey Haggblade

1 Overview
HVX is a convex programming package that solves problems using subgradient methods. There are multiple components, all implemented in the Haskell language. Users of HVX can specify a convex program in such a way that its convexity can be checked by the Haskell compiler itself, before ever running the executable. At runtime, HVX uses symbolic subgradient computations to power its subgradient methods.

We have released our code publicly at this url: https://github.com/chrisnc/hvx/

2 Why Haskell?
At the outset we chose the Haskell language for this project because:

- It has a very powerful type system. It is possible to implement DCP rules at the type level and have the compiler enforce them during typechecking.

- It supports definition of custom infix functions. This allows for lightweight yet fairly readable domain specific languages (DSLs).

- It has an implementation of automatic differentiation (AD) in the form of the Numeric.AD library [Kme14]. AD is a method for evaluating the derivatives of a function. Benefits include being asymptotically faster than finite differences or symbolic subdifferentiation and returning a subgradient function as opposed to a value.

  We hoped to leverage this library to completely avoid writing any subgradient code ourselves. Unfortunately, due to compatibility issues between Numeric.AD and hmatrix, HVX ultimately used a custom symbolic subgradient implementation.

- The hmatrix package provides a simple interface to fast underlying matrix libraries such as BLAS, LAPACK, etc., from Haskell [Rui14]. This allowed us to easily support variables in $\mathbb{R}^n$, expressions in $\mathbb{R}^n$, and functions on $\mathbb{R}^n$. 

3 Example usage

In HVX, a user supplies an objective, constraints, variables, and method-specific parameters to a subgradient method. To solve the optimization problem

$$\begin{align*}
\text{minimize} & \quad \|Ax\|_2 + \|By\|_2 + \|c + x - y\|_2 \\
\text{subject to} & \quad y \preceq 2
\end{align*}$$

the user can write:

```haskell
x = EVar "x" -- declare symbolic variables
y = EVar "y"
tolerance = 1.0e-10 -- U - L < tolerance termination condition
radius = 1.0e10 -- radius of initial ellipsoid sphere centered at origin
ans = ellipsoidMinimize
    (norm 2 (a *~ x) +~ norm 2 (b *~ y) +~ norm 2 (c +~ x +~ neg y))
[y <=~ (EConst $ scale 2.0 $ ones n 1)]
[("x", n), ("y", n)] -- specify the size of the symbolic variables
tolerance radius
```

Figure 1: Example call to the ellipsoid method solver in HVX

4 Disciplined convex programming with type families

HVX implements the disciplined convex programming (DCP) framework [GBY06] within the Haskell type system, so the convexity of a user’s program is checked when the optimization routine is compiled rather than when it is run. This has two benefits: first, it eliminates the performance overhead of parsing the DCP syntax tree when starting up a solver, and second, it prevents users from ever attempting to run programs that are not convex.

HVX defines generalized algebraic data types (GADTs) representing functions and expressions. These algebraic data types combine the convexity type and monotonicity type of the expressions and functions they represent. Function application in HVX is only allowed when both the function and the expression argument have appropriate types according to the DCP composition rules.

These rules are expressed as closed type families on the convexity and monotonicity types [KJeCS10]. A type family is essentially a partial function on types. HVX requires that the result of function applications have valid convexity and monotonicity as determined by these type families. This causes the Haskell compiler to refuse to compile programs that attempt to construct expressions of indeterminate convexity. A snippet of the code that defines the GADTs and type families for functions and expressions is shown in Figure 2.

4.1 DCP with functional dependencies

Another approach that was successful, but which was significantly more verbose, was to use a combination of multi-parameter typeclasses, flexible instances and contexts, and functional dependencies to enumerate the resulting convexity and monotonicity types for function applications.
Like type families, functional dependencies allow output types to be statically determined from input types, but have several limitations. Most importantly, each instance of a typeclass that specifies a functional dependency must not conflict with any other instance. This prevents us from using type variables and pattern matching with precedence, as we do in the type family code in Figure 2. This also has the disadvantage that the convexity and monotonicity information is inherently coupled, as one typeclass instance must account for all of the functional dependencies.

This approach was significantly more verbose and error-prone than using closed type families, but unfortunately closed type families are not available in GHC version 7.6.3, the version that is most widely available from package managers as of this writing.
5 Subdifferentiation

There are a number of different techniques available for subdifferentiating functions from $\mathbb{R}^n \rightarrow \mathbb{R}$. These include using finite differences to approximate a gradient, reverse mode AD and symbolic subgradient computation. Initially AD seemed most attractive, with the Numeric.AD package [Kme14] offering a standard implementation in Haskell. Unfortunately Numeric.AD currently does not support packed vector data types so it is incompatible with efficient matrix packages.

Instead, HVX obtains subdifferentials by computing them symbolically. It recursively applies the subgradient composition rule until a function is broken down into its constituent primitives, whose subgradients are known.

Our implementation of the subgradient composition rule is detailed in Figure 3.

\[
\text{jacobianWrtVar} :: \text{Expr} \; \text{vex} \; \text{mon} \rightarrow \text{Vars} \rightarrow \text{Var} \rightarrow \text{Mat} \\
\text{-- Chain rule: } ddx f(e) = dde f \cdot ddx e \\
\text{jacobianWrtVar} \; (\text{EFun} \; f \; e) \; \text{vars} \; \text{var} = dde_f \vartriangleright> ddx_e \\
\text{where} \\
\quad dde_f = \text{getJacobian} \; f \; \text{val} \quad \text{-- evaluate the hard-coded jacobian of } f \text{ at } e \\
\quad ddx_e = \text{jacobianWrtVar} \; e \; \text{vars} \; \text{var} \quad \text{-- recurse to get the jacobian of } e \\
\quad \text{val} = \text{evaluate} \; e \; \text{vars} \quad \text{-- evaluate } e
\]

Figure 3: The chain rule for subgradients in Haskell

The code snippet in Figure 4 demonstrates how to compute a subgradient manually (note that users do not need to do this, as subgradient computations are all internal).

\[
\begin{align*}
\text{-- declare a symbolic variable} \\
x &= \text{EVar} \; "x" \\
\text{-- give } x \text{ a value (a 4-element column vector)} \\
\text{vars} &= (("x", (4<>1) \left[0, -3, 1, 2\right])) \\
\text{-- define the expression to subdifferentiate: } \text{max}(\text{abs}(x)) \\
\text{myexpr} &= \text{hmax}(\text{habs}(x)) \\
\text{-- compute the subgradient} \\
\text{mysubgrad} &= \text{jacobianWrtVar} \; \text{myexpr} \; \text{vars} \; "x" \\
\text{-- printing mysubgrad gives:} \\
&\begin{bmatrix} 1 \rangle<4 \rangle<4 \rangle<4 \rangle<4 \\
&\left[ 0.0, -1.0, 0.0, 0.0 \right]
\end{bmatrix}
\end{align*}
\]

Figure 4: Computing a subgradient
6 Subgradient methods

HVX currently includes two subgradient methods: the basic subgradient method and the ellipsoid method. The following code snippet shows a portion of our basic subgradient method:

```haskell
subgradLoop :: Int -- Current iteration.
    -> Expr vex mon -- Objective to minimize.
    -> [Constraint] -- Constraints on minimization problem.
    -> (Int -> Double) -- Stepsize function.
    -> Int -- Number of iterations to run.
    -> Vars -- Variables and their current values.
    -> (Vars , Double) -- Optimal variable values and optimal objective value.
subgradLoop itr obj constrs stepFun maxItr vars =
    if itr &ge; maxItr || vars == varsNext
        then (vars , (evaluate obj vars) @@ > (0 ,0))
        else subgradLoop (itr + 1) obj constrs stepFun maxItr varsNext
            where varsNext = map (updateWithSubgrad obj constrs (stepFun itr) vars) vars
```

**Figure 5:** The main loop of the basic subgradient method in Haskell

HVX implements inequality constraints by finding the maximally violated constraint at each step and choosing the subgradient update using that constraint expression rather than the objective if there is a violation.

HVX currently supports only inequality constraints, but equality constraints may be added in the future for at least one of our solvers. For equality constraints in the ellipsoid method, we believe it may be possible to project the initial ellipsoid onto each hyperplane corresponding to the equality constraints at the beginning of the algorithm. We are not yet certain of the practicality of this technique, as there may be numerical stability issues arising from the ellipsoid matrix having a reduced rank.

Note that we support functions of multiple variables through the `Numeric.LinearAlgebra` package [Rui14]. This provides matrix support in Haskell using external numerical libraries like BLAS, LAPACK, and GSL.

7 Acknowledgements

We would like to thank Bryan O’Sullivan and Professor David Mazieres for their help with all things Haskell. We would also like to thank Ahmed Bou-Rabee, Alon Kipnis, Brandon Jones, Ernest Ryu, Jaehyun Park, Linyi Gao, Mainak Chowdhury, Milind Rao, Nicholas Moehle, Professor Stephen Boyd, and Steven Diamond for giving us valuable feedback on our project throughout the quarter. We want to especially acknowledge Jaehyun and his partner Maurizio Calò for their 2011 class project SPY [SPY11], which inspired this project.
References


