HVX: 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. We accomplished this by using generalized abstract data types and closed type families, the latter of which is new feature of the Glasgow Haskell Compiler introduced in version 7.7. 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 What is convex optimization?

Convex optimization is the study of convex programs and methods for solving them efficiently.

A convex function $f : \mathbf{R}^n \to \mathbf{R}$ satisfies $f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y)\ \forall\ \theta \in [0,1]$. The class of convex functions have many desireable properties from the perspective of optimization. For example, if they are differentiable, they attain their global minimum if and only if their gradient is zero. Figure 1 shows an intuitive picture of what it means to be a convex function.

A convex program is a convex objective function, f, a set of convex inequality constraints, $\{g_i\}$, and a set of affine equality constraints. Convex programs are typically written in the form:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g_i(x) \leq 0\ \forall\ i \\ & Ax = b \end{array}$$

3 Example usage

In HVX a user supplies the objective, constraints, and variables of a convex program, along with some method-specific parameters, to a subgradient method. To solve the optimization problem

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

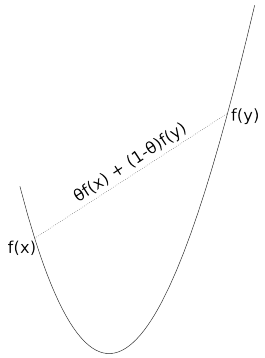

Figure 1: For any $f(x), f(y)$ the straight line from $f(x)$ to $f(y)$ lies above the function f.

the user can run the code listed in Figure 2.

```
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 2: 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 [KJS10]. 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 3.

```
-- GADT for expressions.
data Expr vex mon where
  EConst :: Mat -> Expr Affine Const
  EVar   :: Var -> Expr Affine Nondec
  EFun   :: Fun v1 m1 -> Expr v2 m2 -> Expr vex mon
  EAdd   :: Expr v1 m1 -> Expr v2 m2 -> Expr vex mon

-- GADT for functions.
data Fun vex mon where
  Mul    :: Mat -> Fun Affine Nonmon
  Abs    :: Fun Convex  Nonmon
  Neg    :: Fun Affine  Noninc
  Exp    :: Fun Convex  Nondec
  Log    :: Fun Concave Nondec
-- ... several more of these for different primitives.

-- Type family for the convexity of a function applied to an expression.
-- "newexpr = apply f expr"
type family ApplyVex vf mf ve me where
-- ... 5 more rules here for affine and constant cases.
-- Any function applied to an affine argument keeps its convexity.
  ApplyVex vf      mf     Affine  me    = vf
-- Any non-decreasing function can be applied to an argument of the same convexity.
  ApplyVex v       Nondec v       me    = v
-- Composition rules for non-increasing functions.
  ApplyVex Convex  Noninc Concave me    = Convex
  ApplyVex Concave Noninc Convex  me    = Concave

-- Smart constructor for applying a function to an expression that constrains
-- the output type using the type families
apply :: ( Vex vf, Mon mf, Vex ve, Mon me
         , Vex (ApplyVex vf mf ve me) , Mon (ApplyMon mf me))
         => Fun vf mf -> Expr ve me -> Expr (ApplyVex vf mf ve me) (ApplyMon mf me)
apply = EFun

-- An example of primitive functions that call "apply" for you.
hexp = apply Exp
hlog = apply Log
hexp $ EVar "x"          -- This will typecheck.
hexp $ hlog $ EVar "x"   -- This will not typecheck because convex nondecreasing
                         -- functions cannot be composed with concave ones.
```

Figure 3: DCP primitives and rules with the Haskell type system

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 limitiations. 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 3.

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

Given a convex function f and a point x_0, the subdifferential of f at x_0, denoted $\partial f(x_0)$, is the set of vectors v such that $f(x) - f(x_0) \geq v \cdot (x - x_0)\ \forall\ x \in \mathbf{dom}\, f$. A subgradient of f at x_0 is any vector $v \in \partial f(x_0)$. Figure 6 gives an intuitive picture of another possible chacterization of subgradients. Both optimization methods that HVX implements are crucially dependent on subgradients.

There are a number of different techniques available for subdifferentiating functions from $\mathbf{R}^n \to \mathbf{R}$. These include using finite differences to approximate a gradient, reverse mode automatic differentiation (AD) and symbolic subgradient computation. Initially AD seemed most attractive, with the `Numeric.AD` package offering a standard implementation in Haskell [Kme14]. 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.

For example, one would compute a single subgradient of $f(x) = \sin(|x|)$ symbolically as follows:

1. Use Chain Rule to write $\frac{\partial \sin(|x|)}{\partial x} = \left(\frac{\partial \sin(y)}{\partial y}|_{y=|x|}\right) \frac{\partial |x|}{\partial x}$
2. Compute $\frac{\partial \sin(y)}{\partial y}|_{y=|x|} = \cos(|x|)$.
3. Compute $\frac{\partial |x|}{\partial x} = \text{sign}(x)$.
4. Multiply the above two terms to compute $\frac{\partial \sin(|x|)}{\partial x}(x) = \cos(|x|)\,\text{sign}(x)$.
5. Evaluate $\cos(|x|)\,\text{sign}(x)$ at the given point x.

Our implementation of the subgradient composition rule is detailed in Figure 4.

```
jacobianWrtVar :: Expr vex mon  -- Expression whose jacobian to take.
  -> Vars                       -- Variables appearing in the expression.
  -> Var                        -- Variable to take jacobian with respect to.
  -> Mat                        -- The jacobian.
-- Chain rule: ddx f(e) = dde f * ddx e
jacobianWrtVar (EFun f e) vars var = dde_f <> ddx_e -- matrix multiplication
  where
    dde_f = getJacobian f val         -- evaluate the hard-coded jacobian of f at e
    ddx_e = jacobianWrtVar e vars var -- recurse to get the jacobian of e
    val   = evaluate e vars           -- evaluate e
```

Figure 4: The chain rule for subgradients in Haskell

The code snippet in Figure 5 demonstrates how to compute a subgradient manually (note that users do not need to do this, as subgradient computations are all internal).

```
-- declare a symbolic variable
x = EVar "x"
-- give x a value (a 4-element column vector)
vars = [("x", (4><1) [0,-3,1,2])]
-- define the expression to sudifferentiate: max(abs(x))
myexpr = hmax(habs(x))
-- compute the subgradient
mysubgrad = jacobianWrtVar myexpr vars "x"
-- printing mysubgrad gives:
(1><4)
 [ 0.0, -1.0, 0.0, 0.0 ]
```

Figure 5: Computing a subgradient.

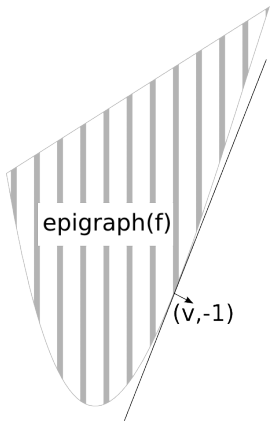

Figure 6: $(v, -1)$ defines a supporting hyperplane for the epigraph of f at x_0.

6 Subgradient methods

Subgradient methods are a class of algorithms for solving convex programs that involve computing a subgradient of objective and constraint functions as a subroutine. The simplest of these methods can be called the "basic" subgradient method, in which every iteration involves a step in the negative subgradient direction of the objective [Boy14b]. The iteration for this method is given by:

$$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}$$

where $x^{(k)}$ is the current value of our optimization variable, $g^{(k)} \in \partial f_0(x^{(k)})$, and α_k is a pre-determined step size specified by the user. Common choices for α_k are constants or diminishing non-summable sequences like c/k for some constant c.

A more elaborate method that also requires subgradients is the ellipsoid method. A detailed presentation of this method is beyond the scope of this report, but can be found in the class notes for EE364B with Professor Stephen Boyd [Boy14a]. Essentially, the ellipsoid method localizes the optimal point of a problem to an (initially very large) ellipsoid. In every iteration, the algorithm uses a subgradient of the objective to cut the ellipsoid through its center, such that the subgradient is normal to the cutting hyperplane. The algorithm then computes the minimum volume ellipsoid containing the half of the previous ellipsoid that lies on the negative subgradient side of the hyperplane. This new ellipsoid always has smaller volume than the previous ellipsoid, and eventually, the ellipsoid is small enough that the algorithm can return the center of the ellipsoid as a nearly optimal point and terminate.

HVX currently includes both the basic subgradient method and the ellipsoid method. The code snippet in Figure 7 shows a portion of our basic subgradient method.

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

```
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 >= 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 7: The main loop of the basic subgradient method in Haskell

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 Mazières 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 Caló for their 2011 class project `SPY` [CP11], which inspired this project.

References

[Boy14a] Stephen Boyd. Ellipsoid Method. In *Notes for EE364b*. Stanford University, May 2014. http://www.stanford.edu/class/ee364b/lectures/ellipsoid_method_notes.pdf.

[Boy14b] Stephen Boyd. Subgradient Methods. In *Notes for EE364b*. Stanford University, May 2014. http://www.stanford.edu/class/ee364b/lectures/ellipsoid_method_notes.pdf.

[CP11] Maurizio Caló and Jaehyun Park. Symbolic Subdifferentiation in Python, 2011. http://www.stanford.edu/~liszt90/spy11.pdf.

[GBY06] Michael Grant, Stephen Boyd, and Yinyu Ye. Disciplined Convex Programming. In L. Liberti and N. Maculan, editors, *Global Optimization: From Theory to Implementation*. Springer, 2006.

[KJS10] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions, May 2010. http://research.microsoft.com/en-us/um/people/simonpj/papers/assoc-types/fun-with-type-funs/typefun.pdf.

[Kme14] Edward Kmett. Numeric.AD. https://github.com/ekmett/ad, May 2014.

[Rui14] Alberto Ruiz. Numeric.LinearAlgebra. https://github.com/albertoruiz/hmatrix, May 2014.