NIELS JOUBERT, CRYSTAL LEMIRE

GETTING STARTED WITH LISZT

AUGUST 2011

This tutorial walks you through the Liszt programming language step-by-step. We assume some familiarity with solving partial differential equations using computational methods on a domain described by a mesh, and a basic understanding of programming languages - variables, functions, types, and control flow. We will introduce Liszt using an example-driven approach, with you following along by writing and extending a small application.

Contents

1 Installing Liszt 5

1.1 Mac OS X 10.5+ 5

1.2 64-bit Debian-based Linux 6

1.3 Testing your installation 6

2 What to expect from the Liszt language 7

3 Example 1: Starting a new Liszt Project 8

3.1 Project Configuration: liszt.cfg 8

3.2 Loading a mesh: mesh.msh 8

3.3 Hello World: Template.scala 8

4 Example 2: Variables, Values, and Type Inference 10

4.1 Liszt Value Types 10

5 Example 3: Topological types, operators and sets 11

5.1 Mesh Topology Operators 12

6 Example 4: The for-comprehension, and applying kernels to sets 13

6.1 Example 4.1: Scalar Reduction Variables 13

6.2 Functions in Liszt 14

6.3 Example 4.2: Objects in Liszt 14

6.4 Special Mesh Element: Exterior Cell 15

6.5 Example 4.3: Boundary Sets 16

7 Vectors and Matrices 17

7.1 Liszt Value Types: Matrix 18

8 Example 5: Fields 19

8.1 Initializing Fields 20

8.2 Field Reads 20

8.3 Writing and Scattering to Fields 20

9 *Example 6: Putting it all together* 22
10 *Viewing our Results in LisztVis (Mac Only)* 23
10.1 *Example 7: Using the 'vis' runtime* 23
10.2 *Example 8: Using watchpoints* 25
10.3 *Example 9: Using filters* 26
11 *Example Project: Scalar Convection* 26
12 *Advanced Topics* 27
12.1 *Traits and Mixins* 27
12.2 *Sparse Matrices* 27
Indices 28
Linear Systems 28
Creating Indices 29
Mixins 30
Non-zeroes 30

1 Installing Liszt

Liszt is distributed as a source package consisting of a compiler, a runtime library and a set of accompanying examples and docs. It requires compilation of the compiler, which in turn requires the dependencies of Liszt to be available on your system.
Liszt depends on:

- gcc-compatible C++ compiler
- OpenMPI,
- *(optional)* CUDA 4.0 SDK if the GPU runtime will be used,
- VTK 5.6 or higher, and
- Scala 2.8.1; which in turn requires
- Java Development Kit 6

1.1 Mac OS X 10.5+

We assume you already have xcode installed on your machine to provide the necessary C++ compiler.

1. **For Lion users:** Install OpenMPI (comes shipped with OS X 10.5 and 10.6)
   (a) Download OpenMPI 1.4 from `http://www.open-mpi.org/`
   (b) Configure: `./configure --disable-mpi-f77 --disable-mpi-f90`
   (c) Install: `make all install`
2. Download **Liszt**, **VTK** and **Scala 2.8.1** from `http://liszt.stanford.edu/`
3. Install the VTK 5.6.1 package
4. Extract the Scala 2.8.1 zip file and place the `scala-2.8.1.final` directory in a convenient location.
5. Add the following line to `~/.bashrc`, filling in the location of `scala-2.8.1.final`, and reopen your terminal:
   `export PATH=$PATH:/WHERE/YOU/PLACED/scala-2.8.1.final/bin`
6. Extract the Liszt file to a convenient location, henceforth referred to as `$LISZT_HOME`.
7. Open a terminal at `$LISZT_HOME\liszt_src` and run `make`
8. Add the following line to `~/.bashrc`, filling in the location of `liszt_src`, and reopen your terminal:
   `export PATH=$PATH:/WHERE/YOU/PLACED/liszt_src/release/bin`

If you want to use the CUDA or MPI runtime, you additionally need to install CUDA 4 or OpenMPI respectively. This can be done without reinstalling anything else.

1.2 64-bit Debian-based Linux

We assume you have the appropriate C++ and Fortran compilers installed. On Ubuntu this can be installed using the `sudo apt-get install build-essential` command.

1. First install the required dependencies using
   `sudo apt-get install csh cmake autoconf automake libblas-dev openjdk-6-jdk`
2. Download **Liszt**, and the **Linux Dependencies Package** from
   `http://liszt.stanford.edu/`
3. Extract the Liszt and Linux Dependencies Packages in convenient directory, hereby referred to as `$LOCALDIR`.
4. Add the following line to your `~/.bashrc`, filling in the location of LOCALDIR, then reopen your terminal:
   `export PATH=$LOCADIR/linux-deps/bin:$PATH`
5. Navigate to the `liszt_src` directory in the extracted Liszt package and open the `Makefile.inc` file for editing. Modify the line defining `VTK_PREFIX` as follows, filling in the location of $LOCALDIR, then save and close the file:
   VTKPREFIX=`$LOCALDIR/liszt-deps`
6. From the same directory, run `make`.
7. Add the following line to `~/.bashrc`, filling in the location of `liszt_src`, and reopen your terminal:
   `export PATH=$PATH:/WHERE/YOU/PLACED/liszt_src/release/bin`

1.3 Testing your installation

Run a quick test now:

· Navigate your terminal to `$LISZT_HOME\liszt_src\tests`

· Run `./run -r single SC`

2 *What to expect from the Liszt language*

Welcome to Liszt, a high performance DSL[1] for mesh-based programming. The Liszt DSL is embedded in the Scala programming language. Liszt inherits many of Scala's features, of which we'd like to introduce the basic concepts now. We will briefly explain these ideas before we dive into Liszt itself.

[1] Domain Specific Language

- A mixed **Procedural** and **Functional** programming style.
- **Variables** and **Values** for mutable and immutable storage.
- **Static Types** with **Type Inference** and **Type Annotations**.
- Integers and Floating Point numbers
- Basic **Control Flow** in the form of `if` and `while` statements.
- **Function Calls** with return values.
- **Singleton Objects** as namespaces for organizing code.

Liszt extends this subset of Scala by adding:

- Data-Parallel Constructs and Semantics.
- Topological types, sets and operators to work with a mesh
- Vectors and Matrices with associated mathematical operators.
- Data storage in terms of Fields

We are interested in your background:

- Who are you?
- What are you working on?
- What languages and libraries do you regularly use for this?
- What are you thinking of implementing today?

3 *Example 1: Starting a new Liszt Project*

A Liszt project consists of a directory containing at least the following three files. **Download our template project at http://liszt.stanford.edu** and extract the tarball to produce a directory containing:

1. `liszt.cfg` - specifying runtime and compiler options.
2. `mesh.msh` - any VTK compatible mesh or Fluent file.
3. `Template.scala` - a source code file, for you to fill in
4. `MeshUtility.scala` - example mesh algorithms

3.1 *Project Configuration: liszt.cfg*

```
{
        "runtimes": ["single"],
        "main-class": "example",
        "mesh-file": "mesh.msh",
        "redirect-to-log": false,
        "num-procs": 1,
        "debug": true,
        "log": "Progress"
}
```

The `runtimes` setting by default specifies the single-core scalar runtime. It can also be one of `mpi`, `smp` or `gpu` to run your code on a different runtime.[2] The `main-class` setting specifies the Liszt **`object`** on which the **`def`** `main()` function resides - the entrypoint of any Liszt execution. The `mesh-file` setting is self-explanatory.

[2] When using the mpi or gpu runtime we compile Liszt using OpenMPI or CUDA, respectively. If you want to use these runtimes, you need to have the appropriate compiler and hardware available. Liszt is compatible only with NVIDIA Fermi-based graphics cards.

3.2 *Loading a mesh: mesh.msh*

Liszt derives its performance by deeply integrating mesh topology with computation, thus it requires loading a mesh as part of the project's configuration. The mesh path is specified in `liszt.cfg`, and accepts all VTK-compatible formats and Fluent File meshes (the .msh file we include is a Fluent mesh).

3.3 *Hello World: Template.scala*

Finally, here is our first Liszt code example:

```
import Liszt.Language._

@lisztcode
object Template {
    def main() {
        Print("Hello world!")
    }
}
```

Let's run this by typing `liszt` at the command line in the project directory. If your path is set up correctly, this will invoke the Liszt compiler, compile the code, and execute the resulting binaries transparently.

The basic elements of any Liszt program can be identified in this example:

- All Liszt code - every variable, constant and function (including main) - is inside an object annotated using the `@lisztcode` annotation. These objects are *singletons* and enable namespaces.
- Each Liszt .scala file starts with `import Liszt.Language._`
- The function `main` is a member of the object specified by `main-class` and takes no arguments.
- Semicolons are optional in Scala.
- Liszt is statically typed, but type inference means we don't have to specify types if it can be inferred.
- To write to standard output, you use `Print(...)`.

4 Example 2: Variables, Values, and Type Inference

Let's extend our example to print multiple outputs:

```
import Liszt.Language._

@lisztcode
object Template {
    def main() {
        val words = "Hello Number "
        var i = 0
        while (i < 10) {
            Print(words, i)
            i += 1
        }
    }
}
```

Liszt, like Scala, supports both constant values such as `words` (**immutables**) and variables such as `i` (**mutables**). `i` can be reassigned, but `words` stays constant after assignment. The type definition syntax varies for constant values or variables as follows, where `T` specifies the type of the variable:

- **`val`** – For constant values, the syntax is as following:
  `val` `a : T = <my code>` [3]
- **`var`** – For variable values, the declaration is as following:
  `var` `a : T = <my code>`

[3] T is the type of a

Notice that we never specified any types in our example! **Type inference** allows us to simplify the syntax further by dropping the type when it can be inferred by the assignment: **`var`** `i = 0` [4]
You *only* need to provide types in the following situations:

[4] `i` is a automatically made a variable with type Integer, since the compiler can infer the type of `i` by the value being assigned to it.

- A variable declaration *without* assigning a value: **`val`** `name : String;`
- All method parameters: **`def`** `foo(amt: Float)`
- Method return values if you explicitly call return, or the inferred type is too general.

Control flow is directly borrowed from scala: **`while`**-loops and **`if`**-statements are the same as in any of the C family of languages.

4.1 Liszt Value Types

The simple types you are familiar with from other languages are "value types" in Liszt - types such as `Int`, `Float`, `Double`, `String`, `Boolean`, `Vec` and `Mat`. Values types are passed and return by value through functions. Value types may be declared as **`val`** (for constant usage) or **`var`** (for variable usage). Later we will introduce Fields - A construct with a Field type is not copied but passed my reference. [5]

[5] Please see the Liszt Language Specification for precise definitions of all the value types.

5 *Example 3: Topological types, operators and sets*

Once Liszt imports your mesh, it makes the mesh topology available through a set of mesh types and functions. Mesh types are always declared as **val**, and are therefore constant. This reflects the current state of Liszt - your mesh topology is static over the course of your program, and accessed through the `mesh` global variable.[6]

[6] Liszt is designed so that our compiler can track how your code uses the mesh, and produce parallel code for different platforms.

Let's extend our hello world example to print information about the mesh:

```
import Liszt.Language._

@lisztcode
object Template {
        def main() {
                val c : Set[Cell] = cells(mesh)
                val f = faces(mesh)
                val e = edges(mesh)
                val v = vertices(mesh)
                Print("Cells: ", size(c))
                Print("Faces: ", size(f))
                Print("Edges: ", size(e))
                Print("Vertices: ", size(v))
        }
}
```

In this example we use four built-in topological operators - `cells()`, `faces()`, `edges()` and `vertices()` - passing the global variable `mesh`, assigning the resulting sets of elements to immutable variables.[7] We then call the `size(s : Set)` function on each of these immutable variables to retrieve an integer size for each set. It is *not possible* to construct mesh elements using integer indexing, they can *only* be accessed using these built-in topological operators.[8]

[7] See the cheat sheet or language reference for a list of topological operators.

[8] As for any rule, there is an exception. See the set of label-based extractors on the cheat sheet, which allows extracting a given topological element from a higher-dimensional element using an integer to specify its relative location in a canonical ordering.

The topological operators in Liszt always returns individual or sets of mesh elements. These mesh elements have associated topological types, on which type inference works as expected:

- `Vertex` — A 0-dimension element, representing a single point on the mesh.
- `Edge` — A 1-dimensional element, connecting two vertices.
- `Face` — A 2-dimensional element (triangle, quadrilateral, etc.), composed of several edges.
- `Cell` — A 3-dimensional element (tetrahedron, hexahedron, etc.), composed of several faces.
- `Mesh` — An entire mesh. Currently the only build-in expression with type `Mesh` is `mesh`.
- `Set[T <: MeshObj]` — A set of vertices, edges, faces, or cells.

5.1 Mesh Topology Operators

Liszt provides built-in functions to access mesh topology, such as `vertices(mesh)`, `edges(mesh)`, `cells(mesh)`, `faces(mesh)` and, for example, `faces(cell)`. Each of these functions are overloaded to take many different topological types - for example, the `cells()` function can be called on `mesh`, or on individual mesh elements. Please see the Liszt Language Specification's section on Mesh Topology Functions for a full list.

6 Example 4: The for-comprehension, and applying kernels to sets

We now introduce our parallel constructs - applying a computational kernel to every element in a set of mesh elements. We will do this by extending our example to print data per mesh element:

```
import Liszt.Language._

@lisztcode
object Template {
    def main() {
        for (c <- cells(mesh)) {
            val id = ID(c)
            val fs = faces(c)
            Print("Cell ", id, " has ", size(fs), " faces");

        }
    }
}
```

This example maps the body (the "kernel") of the **for**-comprehension to each element in the set of mesh cells. **Mapping computation over a set of elements does not guarantee a sequential order, and does not have any loop-carried dependencies. Each cell runs in isolation and potentially in parallel.**[9] The kernel runs once for each element, using different parallel approaches depending on the runtime.

[9] This is one of the most fundamental aspects of the Liszt language, and allows us to distribute elements across many parallel processors.

Things to note:

- `c` inside the **for**-comprehension is of type Cell
- `faces(c)` retrieves the faces of the given cell, returning type `Set[Face]`
- Liszt **does not** have **for** loops - the only looping construct is **while**

6.1 Example 4.1: Scalar Reduction Variables

Lets write our own `size()` function.[10] To count every element inside a set, we need to aggregate data across the **for**-comprehension. Liszt provides *scalar reduction variables* for this purpose:

[10] We now introduce both functions and reductions in Liszt.

```
import Liszt.Language._

@lisztcode
object Template {
    def facecount() : Int = {
        var redvar = 0
        for (c <- faces(mesh)) {
            redvar += 1
        }
        return redvar
    }
    def main() {
        Print("Mesh contains ", facecount(), " faces");
    }
}
```

Scalar reduction variables allow the programmer to use an associative operator across all the elements in a mesh set. Liszt will automatically implement a parallel tree reduction to calculate a final value for the variable in a parallel environment. It is for this reason that **you can only use a single operator on a scalar reduction variable inside the same `for`-comprehension - you cannot read and write it in the same comprehension, you can only use the built-in associative operators.** This code will **not** compile:

```
var redvar = 0
for (c <- faces(mesh)) {
    redvar += 1
    Print("Redvar so far: ", redvar) \\BUG! Cannot read and apply +=
}
```

This example also introduces a function. Note the type annotation on the function declaration: `def countCells() : Int = <body>`

6.2 Functions in Liszt

Liszt programs are built using function calls that takes multiple values and returns a single type. All arguments are passed by value - copied into the function - and returned by value as well. Functions must be declared inside an object. Function definition syntax is:

```
def foo(x1 : T1, ..., xn :Tn) : Tr = <exp>
```

`x1` is an argument variable, `Ti` is the type of argument `xi`, `Tr` is the return type. If `<exp>` returns a value, you precede it by an "`=`". If it returns nothing (the equivalent of `void` in C) you can omit the "`=`".

Main Function Example:

```
@lisztcode
object Template {
    def main() {         // Notice that the ''='' is omitted.
        <liszt code>
    }
}
```

Function with arguments and a return value example:

```
@lisztcode
object MyObject {
    def MyFunction( a: Int, b: Int ) : Boolean = {
        val c = ( a == b )
        return c
    }
}
```

6.3 Example 4.2: Objects in Liszt

Let's modularize our `facecount()` function into a separate module to support good coding style, and memoize it to improve efficiency:

```
import Liszt.Language._

@lisztcode
object MeshCounts {
    var savedcount = 0
    def facecount() : Int = {
        if (savedcount == 0) {
            for (c <- faces(mesh)) {
                savedcount += 1
            }
        }
        return savedcount
    }
}

@lisztcode
object Template {
    def main() {
        Print("Mesh contains ", MeshCounts.facecount(), " faces");
    }
}
```

Objects in Liszt are modules[11]. Objects cannot be nested, and must start with `@lisztcode`. Everything in an object is public, and code outside of functions are executed on initialization.

[11] You can think of modules as a C++ Namespace or a Singleton Class

Things to note:

- `savedcount` is a variable at object level (outside of any function), and can therefore be accessed by other objects and their functions.
- Calling a function or a variable on a different object uses an `<object>.<identifier>` or `<object>.<function>()` syntax.

6.4 *Special Mesh Element: Exterior Cell*

The exterior cell is a special cell element with global id equal to zero. This allows you to find the exterior of the mesh. For example:

```
@lisztcode
object Template {
    for ( f <- faces(mesh) ) {
        val c0 = inside(f)
        val c1 = outside(f)

        if ( ID(c0) == 0 ) {
            // c0 is the exterior cell and c1 is interior
        } else if ( ID(c1) == 0 ) {
            // c1 is the exterior cell and c0 is interior
        } else {
            // c0 and c1 are interior cells
        }
    }
}
```

6.5 *Example 4.3: Boundary Sets*

It is possible that your mesh contains partitions of elements. Boundary sets allow you to access these subsets of the total mesh topology, as defined by the sets in your input mesh file. Boundary sets are not defined inside Liszt - it is already present in the mesh file. Boundary sets are made available as a set alongside the built-in sets such as `edges(mesh)`, as this example shows

```
import Liszt.Language._

@lisztcode
object Template {
    ...
    <MeshCounts code from previous example>
    ...
    val interior = BoundarySet[Face]("default-interior")
    def main() {
        for (f <- interior) {
            Print("Interior face: ", ID(f))
        }
        Print("Mesh contains ", MeshCounts.facecount(), " faces");
        Print("Interior contains", size(interior), " faces");

    }
}
```

Our example mesh happens to contain a set called "default-interior".

Boundary sets must always be declared as **`val`**. The elements contained in the boundary set are identified inside the mesh file using a characteristic string name. They can now be used in the same way as any of the sets returned by the built-in mesh operators. They are generally declared as:

- `BoundarySet[A <: MeshObj]` — `A` is one of `Vertex`, `Edge`, `Face` or `Cell` depending on the definition inside the mesh file.

7 *Vectors and Matrices*

Vectors are fixed-length containers holding a single type of values, and are value types just like `Int` or `Float`. Vectors are defined as:

`Vec[N <: IntM, T]` — A dense vector type of length `N`, where `N` is a meta-integer. `T` is any Liszt value type. Meta-integers are integer literals preceded with underscores. You need to use them to declare vectors and matrices. You may access vectors and matrices with either normal integer values or with meta-integers.[12] For example:

[12] Meta-integers are used when Liszt expects a constant value at compile time, which allows for static checking and optimizations. You should use meta-integers when possible to enable optimizations.

```
@lisztcode
object Template {
    def main() {
        val someVector : Vec[_3,Float] = Vec(0.f, 0.f, 0.f)
        Print("Some vector is: ", someVector)
    }
}
```

`someVector` stores 3 Float values. We can again rely on type inference, and drop the type definition, leading to: **`val`** `myFirstVector = Vec(0.f, 0.f, 0.f).`

Reading an entry in a Vec is done by accessing the desired element by specifying the desired position with a meta-integer, starting from meta-integer _0. Alternatively, the first four elements may be accessed by calling member variables x, y, z and w respectively. A normal integer can be used to access a vector as well.[13] A vector declared as a **`val`** (constant) cannot be modified. On the other hand, a vector declared as a **`var`** can have it's entries **written** to.

[13] This lookup cannot be as well-optimized at compile time, thus we prefer to use meta-integers when possible.

```
@lisztcode
object Template {
    val v = Vec(9,8,7)
    var vv = Vec(1,2,3)

    \\READING:
    Print(v)                                        //Will print (9,8,7)
    Print("(", v.x ,",", v(1) ,",", v(_2) ,")")    //Will print (9,8,7)

    \\WRITING ELEMENTS:
    vv(0) = v(0)
    Print(vv)                                       //Will print (9,2,3)
}
```

We provide pointwise multiply, scalar multiplication, cross product, dot product and normalization functions that work on vectors:

```
def main() {
    val v1 = Vec(0,1,0)
    val v2 = Vec(0,2,3)

    Print(cross(v1,v2))
    Print(dot(v1,v2))
    Print(normalize(v2))
    Print(v1 * v2)        //pointwise multiplication
    Print(5 * v2)         //scalar-vector multiplication
}
```

7.1 Liszt Value Types: Matrix

Matrices are fixed-size 2D storage in Liszt, built from vectors of rows. For example:

```
@lisztcode
object Template {
    val myFirstMatrix : Mat[_2, _3, Float]= Mat(Vec(0.f, 0.f, 0.f), Vec(0.f, 0.f, 0.f))
}
```

`myFirstMatrix` is a 2 rows by 3 columns dense matrix. Each row in the matrix is declared as a Liszt vector of length C.

Access to individual elements

Indexing into a matrix is possible by specifying the desired 2D position using meta-integers or integers. The first element corresponds to the row and the second element specifies the column:

```
@lisztcode
object Template {
    val m = Mat(Vec(1,0), Vec(0, 1))
    val a01 = m(_0,_1)
}
```

A matrix declared as a **var** *may be modified:*

```
@lisztcode
object Template {
    val m = Mat( Vec(1,0), Vec(0,1) )
    m = Mat( Vec(2,3), Vec(0.1) )     // bug! v cannot be modified

    var mm = Mat( Vec(0,0), Vec(0,0) )
    mm(_0,_0) = 1
    mm = Mat( Vec(1,0), Vec(0,1) )
}
```

We provide useful mathematical functions for matrices

Our support is, unfortunately, relatively limited as this is a research language.

```
def main() {
    val v1 = Vec(0,1,0)
    val m1 = Mat( Vec(1, 0, 0), Vec(1, 1, 0), Vec(1, 1, 1) )

    val v2 = Vec(0,2,3)
    Print(m1)
    Print(5 * m1)   //scalar * matrix
    Print(m1 + m1)  //Pointwise addition
    Print(m1 * m1)  //Matrix-Matrix Multiply
    Print(m1 * v1)  //Matrix-Vector Multiple
}
```

8 Example 5: Fields

Our examples so far store global variables and accesses individual elements of a set in parallel. We now introduce the last component necessary to write a real Liszt application - **storing data on mesh elements**, using a construct called `Fields`.[14] Fields map a topological type to a value type. That is, a field stores a specific type of value on each mesh element of a specific type, for example `FieldWithConst[Face, Int](0)` stores an integer on each face. Let's consider a simple example that calculates the center of each face in the mesh:

[14] A field in the Physics sense is a physical quantity associated with each point in a space. Fields in Liszt take on a similar role - associating a value with every instance of a specific type of topology.

```
import Liszt.Language._

@lisztcode
object Geometry {
    val float3_zero = Vec(0.f,0.f,0.f)
    val position = FieldWithLabel[Vertex,Vec[_3,Float]]("position")
    val face_center = FieldWithConst[Face, Vec[_3,Float]](float3_zero)

    def precalculate() {
        for (f <- faces(mesh)) {
            calcFaceCenter(f)
        }
    }

    def calcFaceCenter(f : Face) : Vec[_3,Float] = {
        var center = Vec(0.f,0.f,0.f)
        for(v <- vertices(f)) {
          center += position(v)
        }
        center = center / size(vertices(f))
        face_center(f) = center
    }
}

@lisztcode
object Template {
    def main() {
        Geometry.precalculate()
        for (f <- faces(mesh)) {
            Print("Face center: ", ID(f), " ", Geometry.face_center(f))
        }
    }
}
```

Fields support synchronization and are atomically updated during **for**-comprehensions. They are global data structures that must be declared at object scope. Fields are passed to functions as reference and cannot be returned from functions. A field has to be declared as **val**. However, the stored value T behaves as **var** and therefore can be modified. They are defined as:

- `Field[A <: MeshObj, T]` — A is one of Vertex, Edge, Face or Cell. T is any Liszt value type.

8.1 Initializing Fields

There are two way of initializing fields:

- `FieldWithConst[A, T](<value>)` – Initializes each entry in the field with the value passed as an argument
- `FieldWithLabel[Vertex, Float3](''position'')` – Every element in the field is initialized with a value from the mesh file according to the position of of the vertex.

```
@lisztcode
object Template {
    //each entry set to 0.f:
    val cellField = FieldWithConst[Cell,Float](0.f)

    //entries read from mesh file:
    val positionField = FieldWithLabel[Vertex,Vec[_3,Float]]("position")
}
```

8.2 Field Reads

Fields are indexed by topological element. The return type of a field read is the type stored in the field. For example:

```
@lisztcode
object Template {

    val positionField = FieldWithLabel[Vertex,Vec[_3,Float]]("position")

    for ( v <- vertices(mesh) ) {
        val pos : Vec[_3,Float] = positionField(v)
    }
}
```

8.3 Writing and Scattering to Fields

Fields cannot be written to and read from in the same loop.[15] Like scalar reduction variables, Fields support *associative reductions* inside **for**-comprehensions, but you cannot apply associative operators and read or write the field inside the same **for**-comprehension. If two operators are mutually associative and commutative (eg. + and -), then you can use them both inside the **for**-comprehension.

[15] This restriction allows the Liszt compiler to extract much more parallelism.

```
@lisztcode
object Template {

    val cellField = FieldWithConst[Cell,Int](1)

    for ( c <- cells(mesh) ) {
        cellField(c) += ID(c)     \\valid
        cellField(c) -= ID(c)     \\valid since + and - and associative and commutative
    }

}
```

The following code would produce a compiler error:

```
@lisztcode
object Template {
    val cellField = FieldWithConst[Cell,Int](1)
    for ( c <- cells(mesh) ) {
        cellField(c) += ID(c)
        cellField(c) -= 1
        val r = cellField(c)       //BUG! Cannot change operator type inside a for comprehension
    }
}
```

Fields writes are atomic, so it is safe to write code that has multiple instances of a for-comprehension writing to the same entry in a field. Even if this code runs in parallel, race conditions do not exist and the correct answer is calculated. We call this type of writes "scatters"[16]:

```
val Position = FieldWithLabel[Vertex,Float3]("position")
val Flux = FieldWithConst[Vertex,Float](0.f)

def flux_calc(e : Edge) : Float = {
    return length(position(tail(e)) - position(head(e)))
}

for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val flux = flux_calc(e)
    Flux(v1) += flux
    Flux(v2) -= flux
}
```

[16] The term "scatter" comes from vector operations. When a vector of values needs to be stored into memory, but each element of the vector needs to be stored in an arbitrary location in memory, the vector elements are "scattered" into memory, with multiple elements potentially writing to the same memory location. The wider community has adopted this term to describe writes from a data-parallel kernel that touches memory that does not belong exclusively to the kernel itself.

9 *Example 6: Putting it all together*

Let's write a very simple heat conduction code using Jacobi iteration to solve for temperature across a mesh of rods.

```
{
        "runtimes": ["single"],
        "main-class": "HeatTransferExample",
    ...
}
```

```
import Liszt.Language._

@lisztcode
object HeatTransferExample {
    val rl = 1; val Kq = 0.20f;
    val Position = FieldWithLabel[Vertex,Float3]("position")
    val Temperature = FieldWithConst[Vertex,Float](0.f)
    val Flux = FieldWithConst[Vertex,Float](0.f)
    val Jacobi = FieldWithConst[Vertex,Float](0.f)

    def main() {
        //initialize a single point
        for (v <- vertices(mesh)) {
            if (ID(v) == 1) {
                Temperature(v) = 1000.0f;
            } else {
                Temperature(v) = 0.f;
            }
        }
        //run Jacobi iteration
        var i = 0;
        while (i < 100) {
            for (e <- edges(mesh)) {
                val v1 = head(e)
                val v2 = tail(e)
                val dP = Position(v2) - Position(v1)
                val dT = Temperature(v2) - Temperature(v1)
                val step = 1.0f/(length(dP))
                Flux(v1) += dT*step
                Flux(v2) -= dT*step
                Jacobi(v1) += step
                Jacobi(v2) += step
            }
            for (p <- vertices(mesh)) {
                Temperature(p) += 0.1f*Flux(p)/Jacobi(p)
            }
            for (p <- vertices(mesh)) {
                Flux(p) = 0.f
                Jacobi(p) = 0.f
            }
            i += 1
        }
        for (p <- vertices(mesh)) {
            Print("Temp ", Temperature(p))
        }
    }
}
```

10 *Viewing our Results in LisztVis (Mac Only)*

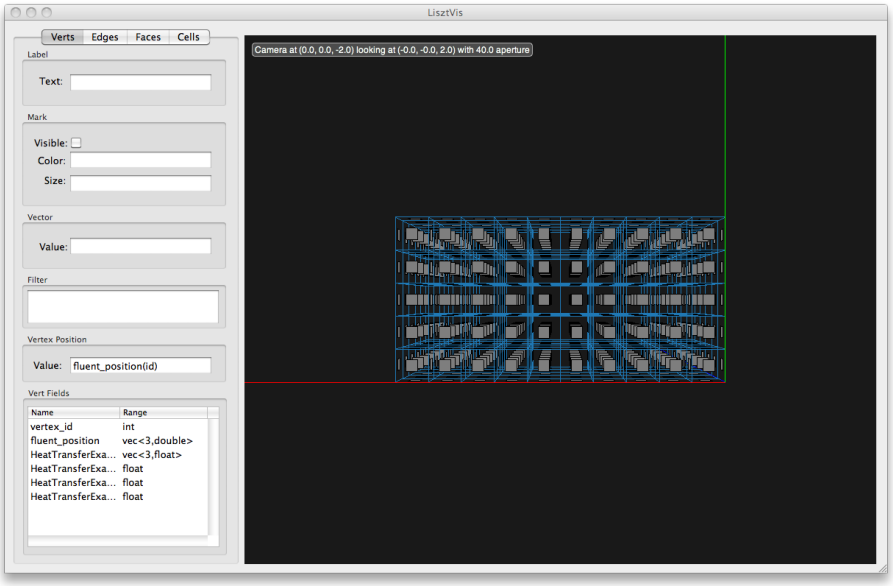
LisztVis is our visual debugging tool. It visualizes the data stored in fields. It further supports watchpointing a field to capture every change in value the field goes through. It further allows visualizing field values on specific topological elements. It is very useful for understanding results and debugging code.

10.1 *Example 7: Using the 'vis' runtime*

Edit the `liszt.cfg` file of the HeatTransferExample code. The `vis` runtime instruments your code to track field state changes. This will hurt performance but enable the extraction of fine-grain debugging data. Alternatively, you can run just `liszt vis` and only see the result of running your code without inspecting field values over time.

```
{
        "runtimes": ["vis"],
        "main-class": "HeatTransferExample",
    ...
}
```

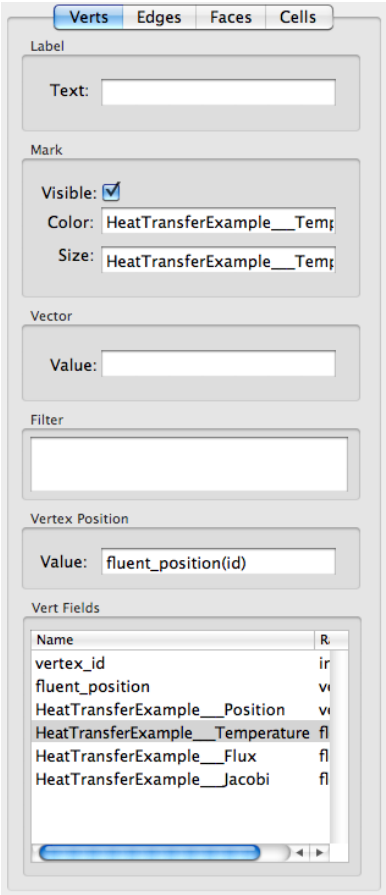
If you now run `liszt` you will see LisztVis opening, as follows:



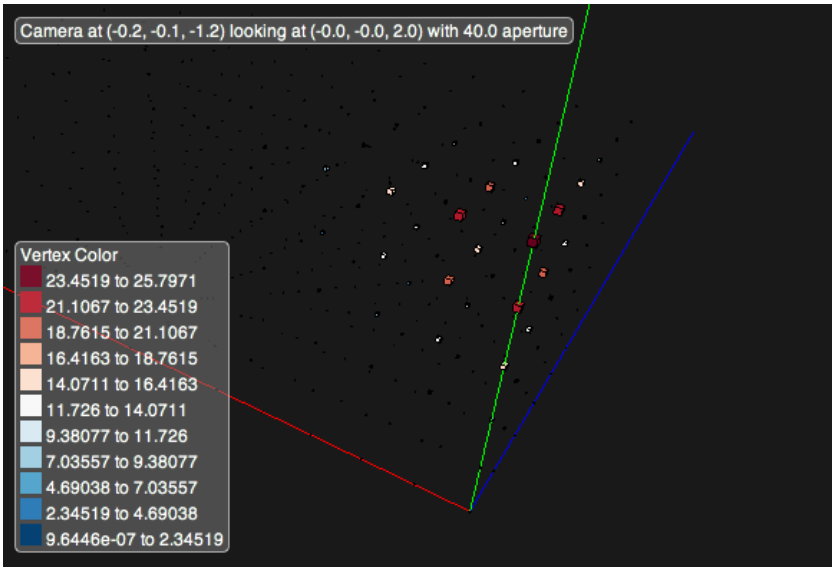
The User Interface is broken up into a control panel on the left and a viewport on the right. The control panel allows us to associate data from our code with the visual elements in the viewport, for each of the mesh types.

Let's visualize the temperature of the vertices:

1. Switch to the *Edges* tab and uncheck *Visible*.
2. Switch to the *Faces* tab and uncheck *Visible*.
3. In the Verts tab:
   (a) Check *visible*.
   (b) Inside *Vert Fields*, widen the *Name* column.
   (c) Drag `HeatTransferExample___Temperature` into the *Color* box.
   (d) Drag `HeatTransferExample___Temperature` into the *Size* box.
4. Navigate through the viewport using your mouse:
   (a) *Left button drag:* Rotate around current center
   (b) *Right button drag:* Translate from center
   (c) *Middle button drag/scroll:* Zoom



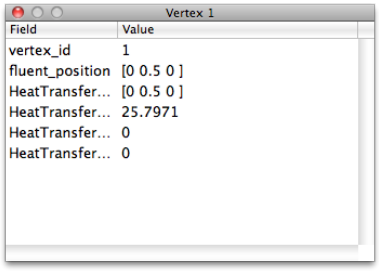
You should now see, after moving your viewport around:

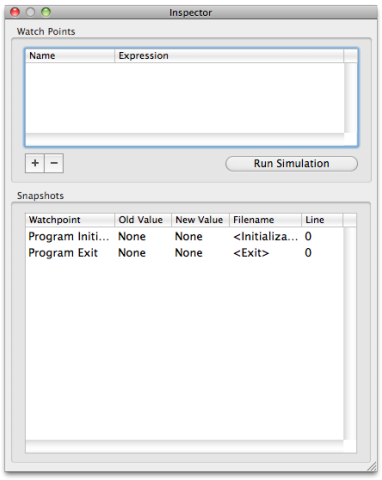


Things to note:

- The control panel allows for adjusting 6 visual elements for each topological element: *Label text*, *Visibility*, *Color*, *Size*, *Vector*, and the *Vertex Position* (vertices only)
- The legend colors are automatically assigned, and shown for the topological type of the currently selected tab.
- Ditto for the *fields* table.

To see more detail about a specific element, click on it to open a detail view of the data associated with it, as shown on the right.



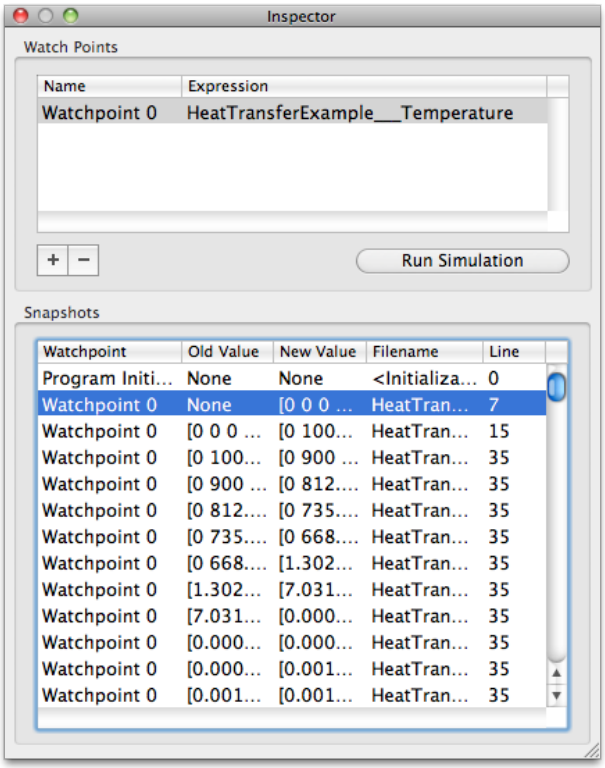


10.2 *Example 8: Using watchpoints*

Since we're using the `vis` runtime we can watchpoint fields to track how its contents change. We use the inspector window, shown on the right, for this.

1. Open the inspector from the menu: `Window > Inspector`
2. Click the + to create a new watchpoint.
3. Into the *Expression* of the new watchpoint, type the name of the field to watchpoint, `HeatTransferExample___Temperature`
4. Click *Run Simulation* to rerun your code, recording the field state every time it changes.
5. After a moment, the list of snapshots appear. Click on a snapshot to select it as the currently active field values. This will change the viewport to display the data at the selected point in time.
6. If you have TextWrangler installed, you can double-click a snapshot and see the code line where the value changed.

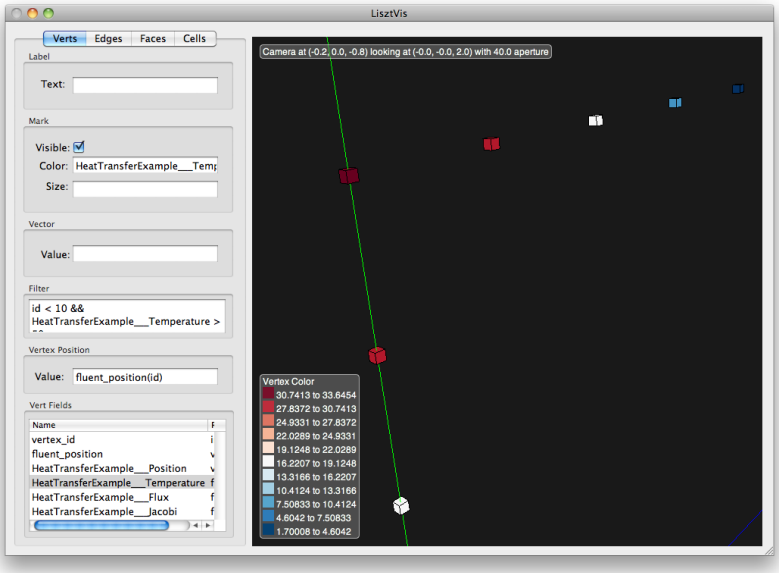
Use your up and down keys to scroll through the snapshots to animate the viewport.



10.3 *Example 9: Using filters*

We now use filters to show a subset of the vertices in the mesh by writing an expression against which vertices are filtered. Say we want to see only the first 10 vertices if their temperature is bigger than 50 Kelvin.

1. Delete the contents of the *Size* box. This is not necessary but improves the visual result for our example.
2. Into the *Filter* box, type `id < 10 && HeatTransferExample___Temperature > 50` and hit enter.
3. As you scroll through the snapshots, you should see only 10 vertices.



11 *Example Project: Scalar Convection*

For a larger example project, see the Scalar Convection code bundled with the compiler at: `liszt_src/examples/scalar_convection`

12 Advanced Topics

12.1 Traits and Mixins

Liszt supports a subset of the trait and mixin interface that scala provides, which lets you reuse common pieces of code by mixing them into your objects. To create a reusable piece of code you can declare a **trait**:

```
@lisztcode trait A {
    var a = 1
}
@lisztcode trait B {
    val b = 2
}

@lisztcode object C extends A with B {
    def main() {
        D.a = 3
        Print(a," ",b," ",D.a) //prints "1 2 3"
    }
}
@lisztcode object D extends A
```

Here we declare two traits `A` and `B` and then mix them into a single object `C`. You can mix the same trait into multiple objects, allowing you to reuse the same code in multiple places, as we have done with `D`. Liszt also allows traits to be parameterized:

```
@lisztcode trait Mult[N <: IntM] {
    def run(a : Float, b : Vec[N,Float]) : Vec[N,Float] = a * b
}

@lisztcode object Mult3 extends Mult[_3]
@lisztcode object Main {
    def main() {
        Print(Mult3.run(Vec(1.f,2.f,3.f))) //prints "[2,4,6]"
    }
}
```

Here the parameter list `[N <: IntM]` indicates that N must be a meta-integer (`IntM`). `Mult3` then supplices `Mult` with the meta-integer `_3` to create an instance of the `Mult` trait for use with vectors of size 3. Parameterized traits allow you to write more general code and use it in multiple places.

12.2 Sparse Matrices

Liszt supports solving linear-systems `Ax = b` defined over the liszt mesh by interfacing with external solvers. This section describes how linear systems are described and solved in liszt.

Indices

To get started, we introduce an abstract data type called Index. An Index refers to a single row or column of a sparse matrix, or it refers to a single entry in a sparse vector. An index is an abstract data type; it not possible to perform integer math on an index. We will introduce a way to construct indicies shortly, but for now let's assume you can obtain indicies and focus on how to use them. Here are some example uses: (assume `i1,i2,i3 ...` are indicies, `v1,v2,v3 ...` are sparse vectors, and `m1,m2,m3 ...` are sparse matrices. Variables are explicitly typed for clarity, but these declarations are optional in Scala).

You can access vectors like so:

```
val f0 : Float = v1(i1)
val f1 : Float = m1(i1,i2)
val index_vector : Vec[_3,Index] = Vec(i3,i4,i5)

//gather a small dense vector from a sparse vector
val f2 : Float3 = v1(index_vector)

val index_vector2 : Vec[_3,Index] = Vec(i6,i7,i8)

//gather a small dense matrix that is the product of a vector of row indicies
//and a vector of column indicies
val f3 : Float3x3 = m1(index_vector,index_vector2)
```

Writing to these objects looks very similar:

```
v1(i1) = 1.f
m1(i1,i2) = 2.f
v1(index_vector) = Vec(1.f,2.f,3.f)
m1(index_vector,index_vector2) = f3
```

Linear Systems

We combine these new types into a linear system, which represents the equation: `Ax = b`. You create a linear system object by mixing in the LinearSystem trait when you want to use sparse matrix solvers.

```
trait LinearSystem {
    type X <: Vector
    type B <: Vector
    type A <: Matrix

    //constructors for the elements of the linear system
    //calling each function returns a new object that can be used in a call to solve
    def x() : X
    def b() : B
    val A() : A {
        type RowIndex = B.Index
        type ColIndex = X.Index
    }

    def solve(A : A, x : X, b : B) //solve for x given A and b
    def nonzeros( non_zero_constructor : (A,X,B) => Any) //method to create non-zeros, see below for detail
}
```

Notice that the row and column indicies are different types, and that `RowIndex = B.Index`, and `ColIndex = X.Index`. This ensures that all accesses to these objects have the right index types. The solve method actually invokes a solver, modifying `x` based on the values in `A` and `b`.

Creating Indices

Let us now revisit the way to construct Index values by first recalling how we might create indices by hand in a non-dsl code. If we had a simple first-order FEM code, our rows and columns of ours sparse matrix map one-to-one with vertices in the mesh. so me might write something like so:

```
type Index = Int
def rowIndex(v : Vertex) : Index = ID(v)
def colIndex(v : Vertex) : Index = ID(v)
```

A more complicated code might use linear transformations of the ids of mesh elements to achieve the same effect, mapping mesh topology to integer indicies.

In Liszt, we retain the concept of mappings from mesh topology to indicies, but we make the mappings abstract. In liszt, you write:

```
@lisztcode
object MyLinearSystem1 extends LinearSystem {
    def rowIndex(v : Vertex) : A.RowIndex = AutoIndex
    def colIndex(v : Vertex) : A.ColIndex = AutoIndex
}
```

Instead of implementing these methods using linear transormation of IDs, Liszt will automatically implement these methods. Liszt will assume that for each method, and for each set of unique arguments to the method, a unique Index object should be created. Let's consider a more complicated 2nd order FEM case where indicies exists for edges and vertices. Here we just add additional methods to return indicies for the edges:

```
@lisztcode
object MyLinearSystem2 extends LinearSystem {
    def rowIndex(v : Vertex) : A.RowIndex = AutoIndex
    def rowIndex(e : Edge) : A.RowIndex = AutoIndex

    def colIndex(v : Vertex) : A.ColIndex = AutoIndex
    def colIndex(e : Edge) : A.ColIndex = AutoIndex
}
```

You can also return `Vec[N,Index]` objects, which are useful if you are solving a vector field:

```
@lisztcode
object MyLinearSystem1Vec extends LinearSystem {
    def rowIndex(v : Vertex) : Vec[_3,A.RowIndex] = AutoIndex
    def colIndex(v : Vertex) : Vec[_3,A.ColIndex] = AutoIndex
}
```

The arguments to these methods must be mesh topology, and the return types must be indicies.

Mixins

We realize that declaring these mappings from topology to index is tedious and verbose, especially in the simple FEM cases. To work around this, we provide mixin traits for the most common cases that already are set up correctly.[17] You can think of these mixins as already pre-built recipies that you can pick and choose when you need. Here is an example for first-order FEM on triangles:

[17] This library is still in development and is not yet complete

```
trait Trangle1 extends LinearSystem {
    def rowIndex(v : Vertex) : A.RowIndex = AutoIndex
    def colIndex(v : Vertex) : A.ColIndex = AutoIndex

    def rowIndices(f : Face) = Vec( rowIndex(vertex(f,0)),
                                    rowIndex(vertex(f,1)),
                                    rowIndex(vertex(f,2)))
    def colIndices(f : Face) = Vec( colIndex(vertex(f,0)),
                                    colIndex(vertex(f,1)),
                                    colIndex(vertex(f,2)))

    def triangle1Nonzeros() //allocates non-zeros for first-order triangular FEM
}
```

Notice that we also provide two helper functions rowIndicies, and colIndicies that produce vectors of indicies for the canonical triangluar element. Given a triangle "f" and a linear system "l" you might use them like so:

```
    //declare linear system
    @lisztcode
    object LS extends Triangle1
    object Main {
        val A = LS.A()
        for(f <- faces(mesh) {
            //gather dense matrix from sparse matrix
            val matrix : Float3x3 = A(LS.rowIndicies(f),LS.colIndicies(f))
            //update dense matrix locally
            performUpdate(matrix)
            //scatter dense matrix back into sparse matrix
            A(LS.rowIndicies(f),LS.colIndicies(f)) += matrix
        }
    }
```

We provide traits for the most common FEM types, so for the most part simple codes will only need to call a few functions to access the matrix.

Non-zeroes

Finally, we need a way to declare where non-zero entries are in these objects. Declaring non-zero entries explicitly ensures that the solvers are always working with the minimum number of non-zero entries.

Otherwise we would need to rely on compiler analysis which may be overly conservative and lead to slower solvers.

For this we provide the nonzeroes() method in LinearSystem. This is called when you want to change the layout of non-zeroes in the LinearSystem. For 1st order FEM , you might call it like so:

```
l.nonzeroes {
nz =>
    for(v <- vertices) {
        nz.x(l.colIndex(v))
        nz.b(l.rowIndex(v))
    }
    for(f <- mesh) {
        for(v1 <- vertices(f)) {
            for(v2 <- vertices(f)) {
                nz.A(l.rowIndex(v1),l.rowIndex(v2))
            }
        }
    }
}
```

Any entry referenced in the block given to nonzeroes will be presumed to be non-zero. LinearSystems are allocated assuming that all values are zero. You must first call nonzeroes before using the linear system. When nonzeroes is called, all old non-zero values in any object derived from the linear system are invalidated. This allows you to change the format of the matrix later on in the program (if, for instance, you want to change the order of some elements). For the common FEM cases, an already implemented nonzeroes helper method will be provided in the mixin trait, so you will only need to call it once to initialize the matrix.