PROFESSOR: Right, so now I'm going to walk you through some examples. And just so you know where this stuff is-- so you all have your LLGrid accounts. And in your LLGrid accounts-- and I'm looking over here because these people are going to check to make sure I don't say anything wrong. So this is, by the way, this is [INAUDIBLE] over here and Julie Mullin.

They are our expert consultants, PhDs in computational science, that help you all. And we're all eternally grateful to them for helping us get all this technology to work. So if you go to your LLGrid account, there should be a link in there called Tools. This is where all the software is that we provide as part of LLGrid.

And there'll be one called d4m_api in there. So there'll be a folder just like this. And just so you know-- so all the lectures are there for you.

And I am putting them all in for public release. And all the software we're going to post on the internet too. And so just make it easier for you to use, use with your government-- use with your sponsors, use with your projects, all that kind of stuff. We try and do that so that people aren't constantly asking us, well, what can I share?

It's like, I'm going to put it on the internet. So there you go. We want to focus on the examples directory here.

The order of the examples is numerical order. So we are going to go through examples in the folder called one and then two and then three. And this really corresponds to kind of the first three lectures, then the next three lectures, then the next three lectures, OK? And so today, we're going to go in here to this first one.

And then here is kind of like the first three of those three. We're going to go and we're going to review this set here, OK? So these are the-- the examples of have this sort of-- they basically take the first two letters of the folder-- in this case, [INAUDIBLE] intro AI 1234. Those are the actual examples.

If you see other files in there, those are supporting files. You don't run those directly. So I'm going to start my-- I'm going to-- I always run MATLAB from the shell. Other people run it from the IDE.

I'm going to create a shell here. [INAUDIBLE] folder. I'm going to start my MATLAB up. This will take a minute because we're reusing 2012B, which is a little bit slower on this computer. All right, there we go. And you'll be pleased to know I develop all the [? D form ?] software on this little computer.

And people are going, well, why don't you get a big giant workstation to develop your programs on? It's like, well, because if it feels OK for me on this computer, I know that most of you have much better computers. And it will feel-- should feel very, very, very good for you.

All right, so we're in the folder. And I'm just going to run the first program-- AI1_SetupTEST. Actually, before we even get into that, the first thing you're going to want to do is check before you run this that your D4M code has been properly set up. The simplest way to do that is when you start MATLAB, if you type help D4M and you get a list of all the functions in D4M, then that means that your path is set up properly.

And this is just a list of all the different functions. We break them down in different types. You've got a little sort of like how to set up here, although a lot of this stuff isn't-- is written for more people on the outside world. And we go through all the functions and we categorize the functions. You know, some are functions that we really expect to use all the time.

Some are functions that you might use once in kind of rare instances. And then we also have functions like, look, you really-- they're there. They're in the library.

These are really not meant for you to use, though. They're really internal supporting functions. But they're there in the library.

All right, so that's the first thing you want to check is to make sure that's set up. If you have any issues, send email to grid-help@ll.mit.edu and the people will help you in and check it out. And don't be surprised if some of you do have an issue. This is the first time we've really rolled it out to such a large audience. And so we absolutely expect people to have little things that will pop up.

All right, so I'm going to run the first example here. I should say we also have tested all the stuff. For those of you who are utterly averse to commercial software, depending on your

religious preferences, we also run with GNU Octave, which is the GPL version for those of you who refuse to run non-free software. And so this-- all this stuff should also work with that as well just for those people who prefer to use that type of environment.

But generally, MATLAB-- you know, it's pretty available here. And so we certainly encourage you to use that. So we're going to run the first test here. AI1_SetupTEST-- Go. Yay, it worked.

It's embarrassing when you're recording and these don't work. So I'm going to walk you through some really rudimentary stuff here in this example. So one of the things in D4M that you're dealing with a lot is lists of strings-- long list of strings, millions of distinct strings.

Now, MATLAB has data structures that do support lists of strings, cell arrays being one of them. There's other data structure that you can use to do that. They naturally support these.

But they tend to be very memory intensive and very slow. And so given our whole thing here is we have a real focus on performance, me giving you a great tool that's 1,000 times slower than other techniques is not very helpful. So we are going to be dealing with list of strings all the time. And so in D4M, a list of strings is a row vector of characters, OK? I've highlighted here.

And the last character in the row vector is the delimiter. In this case, it's the comma. You're at the end. That is the delimiter.

It can be whatever you want it to be. It can be semicolon. It can be a dash. It can be a space.

I tend to recommend new line, which is ASCII character 10-- very safe delimiter. But whatever that last character is, that's the limiter. And you could have different lists, different strings with different delimiters.

It should handle those situations just fine. But within a list of strings, it needs to be that-- that last character will be the delimiter. So I'm actually creating here a list.

I'm going to creating a set of triples here-- r, which is the rows, c, which is the columns, v, which is the values. And so I have an r, which is this. I have a set of c here, which is this. Vector here and then a list of values-- and the values in this case are just appending the row and the column together with a dash.

And all three of these use the comma as-- it's the last character-- as the delimiter, all right?

And now I'm going to create an associative array, which is the fundamental data structure in D4M. It's what allows us to bridge linear algebra and strings together.

So this is a constructor command for an associative array-- so Assoc. And then we give it a list of row keys. These are called-- we often call them keys-- list of column keys and a list of string values. And this construct the associative array a.

And in MATLAB, since I haven't terminated this with a semicolon, it will now will print out the list of triples. So now you can see the list of triples we can construct here was row a, column a, value a-a, row aa, column a, value aa-a, and I won't read the whole list. Yes, Darryl.

AUDIENCE: Question. [INAUDIBLE]

PROFESSOR: That is an artifact of this printing out display. It will show you the delimiter that you actually used here just as-- and it's good to know that you--

AUDIENCE: It's not really there.

PROFESSOR: It's not really there. Yeah, I mean, it's there and it's not there. But it's showing you the delimiter.

And after a while, you just kind of learn to ignore that. The only time this does become a little bit of an issue is that that is the newline character. Then the formatting gets a little bit-- and we've actually built in routines that allow you to take an associative array, replace the delimiter with something nicer in one command so that you can print it out and doesn't look so crazy if you're using a new line. Typically, though, you tend to be doing this print out command on small things.

So it's fine. So that's the whole list, you see there, of all the different entries. If you use the disp command, it will actually display the internal structure of the associative array. And an associative array object in D4M has four fields total.

That's it. We do everything with just four fields. And the four fields are a set of row keys.

This is a lexicographically sorted list of the unique row keys. And it's stored as a string list. So in this case, we had six unique row keys. And we have six entries in this row string list.

Likewise, the column is the same thing. We have these six unique entries. And you see they inherit the delimiter that was passed into them.

If I'd used different ones, you would see different delimiters. And then the value, which is another list that shows all the different strings, OK, and then a matrix, which shows-- which is a six by six matrix. So this is a six by six associative array.

And this is the pointer. Basically, the value stored in a six by six sparse matrix points to the index of the value. So you can view that as a pointer to the values. And this is how we can have values that are actually strings.

That may not be quite clear. There's an easier way to do this. So we have a little routine function here called displayFull, which produces a nice tabular view of the data.

So here's the row keys, the column keys, and the values. And you see this was the matrix I constructed. You had a full first column and a full first row and then values along the diagonal. And then what we're going to do is we want to save this.

We want to write it out. So we're going to write it out to-- we have a function [INAUDIBLE] assoc to CSV. So I pass an associative array. I give it the row terminator and the column separator and a file name and we'll write that data out to a CSV file, which is very convenient.

[INAUDIBLE] actually hide that. You can see here is the CSV file. So you can look at it. There's the CSV file. You can zoom in on that for you. Can you see?

There we go. Those are our six rows, six columns against first row, against first column, and our diagonal, all right? All right, so let's go back here. Now let's go to our next example, which is going to be AI2.

And now we're going to talk about how-- we sort of described how we put data into an associative array. We're now going to talk about how we query it or get data out of the associative array. And one thing that's very nice about the queries that I will show you as we get to the later parts of the [? course ?] we do databases, whether you're querying an associate array that's in memory or binding to a table, it's the same.

We try and make it so that almost everything you would do in an associate array, you could also do on a table. So you can write your programs in associative arrays and then switch a couple things and now it should also work on tables in the database the same way. And so we try and preserve that concept.

The only difference is that a table [? in ?] databases just can be much, much bigger than an associative array you would have in your memory space. All right, so you're going to run that. That's the next example.

So the first thing we did is when we wrote out the associate array, we're going to read it back in. So we have a nice function here called ReadCSV that reads CSV files. Just so you know, Microsoft Excel does write out a non-standard CSV file.

Microsoft Excel [INAUDIBLE] common-- that is, if you have a row that is empty after a certain point, it won't write out those commas, which is technically probably not conformant with the official CSV format, although I don't know there is a sufficiently written down CSV format, right? You can describe it in one line. So you just have to be careful about that.

If you see this issue, people write out a CSV from Excel. And if there's an empty last-- if it's last row is not fully dense or if its last column is not a dense column, you can get this issue and it will screw this stuff up. We also don't support quoted strings. Way to do that, though, is to create a TSV file.

So put tabs in there. And we support tab-- just, if you just call this a TSV file, when you write it out, you can have your separator via tab instead of column, a comma, and then you're all good. And so that's how we support that.

So we're going to now-- so that's how you read it in. And now we're going to do a whole bunch of different queries. These are all what are relatively complicated queries. So the first one here is get me rows a and b.

So if I pass in a string list just like the-- whoops, didn't want to do that. Let's see here. We pass in a string list of the same type that I have before.

And I just say this exact same type of indexing that we normally have in MATLAB where you're like, I want to get it a set of rows. I give it a set of rows. And so this says, get me rows a and b. And give me the whole row-- colon-- use the standard MATLAB syntax that colon means full row. Likewise, if a was a binding to a table, it would deliver the exact same query to the database and return the exact same thing in associative array.

Here's another more complicated that says get me all rows containing a. So our wild card character is a little liberal here. It will just give you anything containing an a. It doesn't really respect beginnings or endings or anything like that. It's kind of a regular expression.

And this says get any row containing a and column's one through three. So we can use numerical indexes. Sometimes, you're just like, I don't care what the row keys are.

Just give me the first 10 columns or the first 10 rows. And that will return this, OK? This is one feature that does not work on all databases.

Some database have a concept-- the numerical index of a column. And some databases do not. So it's dependent on the actual database.

Here's another one. This is a range query. So if you remember in MATLAB, if you do colon and two values, it gives you a range. So we can do a range query here.

So a-- give me all columns a through b. If you really wanted to just get-- we have something called a starts with, which is something like-- a lot of times, you'll be like, I want to just get the rows or the columns that begin with a certain string. So we have a little shorthand routine here that constructs that for you.

So this says, get me all rows starting with a and c. Likewise, I can do the same thing with columns. I can say get me columns a and b. I can say get me all columns that contain a with columns-- with rows one through three.

Like I said, I can do column ranges here. I can say, give me all columns a to b. Likewise, I can do the starts with command as well as give me all columns starting with a or c. And then finally, I think this kind of fun, we can actually query the values.

Again, this is a feature that's not in-- [? it's ?] supported with the database. But it is supported with the associate array, which is if I say get me all-- return an associative array were all the values are greater than b-- I'm sorry, less than b, OK? So let me just show you what that looks like. So this was that query.

And this is what it looks like. So I do display a-- and you see now we don't have any values that begin with a b-- all the values, OK? Another thing that we should notice here is that this is now a three by three matrix, not a six by six matrix.

Associative arrays never store an empty row or an empty column. That is a big difference between traditional sparse linear algebra and associative arrays. In sparse linear algebra, you could have a row of all zero-- an empty row or an empty column, not an associative arrays.

Associative arrays, you either have-- there's going to be-- if you have a row or column, it's going to have an [? entry-- ?] yes.

AUDIENCE: [INAUDIBLE]

PROFESSOR: So basically, all it does-- so it's basically-- so if we had a value to begin with, c, that would also not be included. So it's basically-- lexographically, we compare the value aa with b. And we say, is it lexographically before b? If so, it satisfies the condition.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yes, yes. So that's the policy that we [? made-- ?] so strictly, the algebra of associate arrays doesn't require lexographical ordering. That's an implementation fact. It's a very important implementation fact. We are constantly maintaining lexicographical order inside the data structure. Yeah, Darryl?

AUDIENCE: [INAUDIBLE]

PROFESSOR: That's right, yeah-- three by six, sorry, three by six. Yes, yes-- because these are full here, right? But you see that the ones-- none of the ones with b in it happened because they didn't satisfy criteria.

So therefore, they're empty. And so therefore-- so just very important thing to know. All right, moving on here to the next--

AUDIENCE: [INAUDIBLE]

PROFESSOR: You define other [? orderings. ?] The mathematics would no doubt admit that. It's really backed in, though, into the implementation.

I mean, because we rely on the MATLAB sort command-- and as far as I know, that does not allow you to have other orderings. Now, you could obviously do rehashed. You could hash your rows and keys to some other thing and then have an associate array that maps those back and forth.

And in fact, people do that all the time and just have-- and you could do that. But that's how-- and if you really, really want the order-- the main thing, though, is to kind of think of it, though, is that the ordering doesn't really matter. It's really a device that allows me to do fast lookups. And you shouldn't really care about the ordering.

So let's move on to the next example. So actually-- so we're going to do AI3. So now we're going to do some math on this stuff, OK? So here we go.

All right, so once again, I read in my data that I constructed in the first example. The values of that data are strings. But if I want to do math, sometimes I don't want to do math on strings.

Sometimes, I want to do math on numbers. So in D4M, the associate arrays can also be numbers, which can be very convenient for doing mathematical operations. And so what we have here is we have this command called dblLogi, which is a shorthand for-- and it's kind of cut off here-- applying logical to the associate array, which takes all the values and converts them just to a zero or a one-- basically throws away the strings. Or if it was a numeric value, just [INAUDIBLE] are you there?

You're a one. You're not there, you're a zero or you're empty. And then since we can't do arithmetic on logicals in MATLAB, we have to bump them back into doubles.

So we do this so often, we've actually made a little shorthand here where we call dblLogi. And you'll see that all the time in the class because I just don't like to type all the characters [INAUDIBLE] writing double logical gets very redundant. Now, if we go and look at that data again, we see when you do displayFull that instead of where before we had values of these various strings, we just now have numbers-- just ones.

So that's convenient. Now I can do arithmetic on them. So let me do the first thing here. One of the things I might want to do is sum all the rows-- so the MATLAB sum command.

It's basically-- you're giving it essentially the dimension to eliminate. The first dimension we want to eliminate or sum over is the row dimensions. So that means it's summing up all these rows.

It's basically squishing the matrix and summing them up. And now we have a new associative array, which is essentially a one by six associative array. You see now that the row key is empty.

Because when we sum, the definitions of the rows sort of kind of go away. So we don't-- you don't have to have strings to be your row keys. You can have numeric row keys. Just leave the row entry empty.

But there are some cautions with that as well. So when we sum that and you see [INAUDIBLE] we have all the columns, the values, because they're not strings, are just stored in this a matrix itself. And you see that we had six and two and two and two and two and two. So when we sum these up, that's exactly what you would expect.

Moving along, we can do the columns. So here I'm going to sum. And then I'm going to do display full, which is the same as just kind of listing it. And we see here we now sum the-- compressed all the columns.

So we have a column vector. And this shows the row labels of that column vector. We now have a new column label, which is just one. And then you see the actual values there-- again, a very useful thing.

People do this all the time summing of their rows and columns. Let's do a simple join. So I'm going to say give me a column vector A. Get me another column vector b, right? And now I'm going to join these two together.

Now, I could just do aa and ab. But I'd get an empty matrix. The reason is because it would attempt to do the joins and they would have different column labels. And so when we do joins, we're intersecting the two sets of row and column keys together.

And if they have sep-- those are just separate columns. They add them together, they have no intersection. However, if we have this function called no call, which is actually blows away the column and basically gives them all, in this case a column value one, we can now add them together. And we can actually find where these two things have a common value. So that's a fairly simple way to do a join.

This is something called a facet search, which basically says, all right, I'm going to join these things together. So I'm going to create a column vector. But then I'm going to transpose that over here. So I transpose it. And then I'm going to multiply it back with the original matrix.

This gives me a count of essentially all-- given all columns, all rows that contained-- had an entry in column a and B, can you now sum up their rows? And this is a fairly-- this is actually the mathematical basis of, if you ever do in Google search it does auto-find, it's essentially trying to do this type of operation. It's trying to guess what the next most popular topic would be. And this is essentially-- this mathematical operation does that kind of thing.

We actually have a bunch of applications in lab that use this quite heavily. And I can display

the transpose of that-- so to make it a column vector. And as you see here, we have a bunch of columns here and then values.

And then we can actually do things like sum and normalize. So we can divide. So I'm going to normalize them. You can display that.

You see now you get the probabilities associated with these things. Just basically, you can just kind of go on your way doing math. This shows essentially the correlation of the columns a and b. But why not just do all the correlations at once?

So I'll focus on here. So we have this function square in, which is the same as a transpose a but a little bit faster if you're just squaring something with itself. I'm going to get the diagonal of that. So if I use this function Adj, which stands for adjacency, it'll just pop out that a matrix by itself.

Whenever you use Adj, you're getting a straight sparse MATLAB matrix. And you can do any operation on that that you want that MATLAB supports. So it's a really-- I highly recommend that. If you can't figure out how to do it with associate arrays [INAUDIBLE] just pop up that adjacency matrix.

Do whatever you want. And then you can just-- as long as you didn't change the size, you can just stuff it right back in at essentially no cost. So this is basically copies that field out.

And we're going to get the diagonal. The reason we want to get the diagonal is when we do correlations, we always get this dense diagonal. I want to eliminate that.

So I'm going to then now take the adjacency matrix of the original correlation matrix, subtract the diagonal. And then we have this function [? putAdj, ?] which is kind of the inverse of Adj, which just says I have an associative array. Replace that a with this. No checking, instantaneous copy, very fast, but make sure it's the right size. If it's the wrong size or you've reoriented the columns in some way or the rows in some way, then it won't make any sense.

So it's a little bit of an advanced feature, but it's a very powerful one. You can pop stuff in and out of these structures very quickly. And that way, if you ever need to do math that we don't support, you can just do that directly. And so if we look at that, we see what you would expect-- a nice correlation matrix here, symmetric with no diagonal and the counts of each one.

Right here-- so let me move on to the next example here, which is AI4. And this just shows you

different ways to construct associative arrays, some of the various kind of more degenerate cases. So I have a bench [INAUDIBLE] creating a string here and a bunch of numeric values.

The point here is when you construct an associative array, one, we have full support for all these degenerate empty conditions. You give me anything with any kind of empty, we're going to return an empty associative array. MATLAB has outstanding support for empty objects. You can just keep on passing.

You don't have to-- we basically-- what I'm saying is we do a lot of checking. If the thing is empty, short circuit, return an empty type of thing. So you're not having to constantly check if something is empty or not in order to proceed. So that's what we do there.

That's just showing you all these [? supported. ?] In addition, you can do mixed types. So when you construct an associative array, if one of the values is a scalar-- like, for instance, you can have that.

You don't-- you know normally, as in the MATLAB sparse constructor, when you give it a set of triples, they all have to be the same length. Or any one of them can-- or any of them can be scalars. So this just says, I have a bunch of columns strings, a bunch of numeric values.

Oh and by the way, they all have the same row. So this is a very quick way to construct a row vector. I don't have to go and replicate this a that many times to make this work.

Likewise here, I could have a numeric value for the actual rows, a string value for the columns. And I want everyone to have the same value. They're all going to have the value of a. And you can do variations on that theme.

So you do-- these are fast ways to create row columns, row vectors, or column vectors or values with all or constant numeric value and other types of things like that. And then we can just display one of these. And you see here-- I think it was which one we displayed-- this first one, which is a row. And you can see when you display that, it just has one value for the row.

These are the different columns. These are the different values. Then again, if we display full, you see in the tabular form that's what you have.

So that brings us to the end of the first lecture. I'm happy to stay for questions that people might have. Please go and check out your LLGrid account.

Copy the examples. Don't try and work out of the-- you'll get, like, permission denied errors and stuff like that. Copy the examples to your home directory.

Try and get your D4M working. Try and run just this first example. Make sure it behaves the same way that we had here.

And then send email to grid-help. And then as we get into the next week, we'll do homeworks that are a little bit more substantive. But this is just to make sure the technology is working for you. And that's-- we'll wrap it up there. Thank you.