1. We want to understand how the brain does things
2. This is important, not only for clinical applications—that is treatment of various brain disorders—but also because we want to understand the various algorithms that allow the brain to process such large amounts of information and react to the environment.
3. Now, to give you an idea of the size of the challenge: the most recent estimate puts the number of neurons in the human brain at 86B.
4. These are connected to each other with 100T synapses
5. Recently, we've also started to understand, rather realise, the importance of support cells—the glia—85B of them.

1. Now, experiments of course, provide us with direct evidence about the brain.
2. fMRI, EEG, to voltage and calcium imaging, patch clamp recordings
3. Now: large scale brain observatories
4. Models/theory are necessary for:
5. combining independent experimental results into unified theories
6. exploring these complex systems across wider range of conditions
7. generating new testable hypotheses

1. RNNs are appropriate for lots of projects, for example.
2. So are whole brain neural mass models.
3. But, to really understand the underlying mechanisms that give rise to emergent behaviour, we must model the brain at biophysically detailed levels.

1. Ideally, what we want is for all the stages to be connected seamlessly, but this is not true in practice.
2. We create our model, ideally re-using already published components.
3. Then before we simulate our model, we want to validate it in some way.
4. We also want to analyse and visualise our model description before.
5. Then we iteratively simulate and fit our model to data, or to produce a certain behaviour.
6. Finally, we want to publish and openly share the model so others can use it in the future.

1. There are a lot of software tools out there for users to pick from, for different levels of modelling, optimisation, analysis.
2. For each stage.
3. But, they aren’t designed to work together.
4. They have their own designs, their own APIs, syntax, model representation, and usually their own suite of custom utilities to work with their model representation.

1. This means that for example, a model written in simulator A, say NEURON, cannot just be re-used in another simulator.
2. In fact, because a majority of these tools do not have a well defined model description, even re-using models developed in the same simulator can be quite hard.
3. It takes a lot of human resources to translate/convert models to be able to re-use them.
4. It also makes it very hard to study or analyse these models.

1. Now, this isn't a problem unique to computational neuroscience, or even neuroscience.
2. Multiple scientific fields have run into this issue, and the answer that they've all come up with is to standardise.
3. Standards allow the representation of data and models in specific, agreed formats.
4. Once a standard is agreed upon, everyone can target it—tools, representations, utilities.
5. If one knows what the data is going to look like, one can then develop tools and APIs around it.
6. And instead of everyone writing a tool for their own standard, every tool anyone writes for the one standard can be used with everyone's data.

1. The idea being that by being the standard, various tools that support various stages of the model life cycle can then work together.

1. It consists of two components. The specification or the standard, and the software that adhere to this specification.

1. The standard itself has two different components.
2. There's the schema, which formally specifies the model description—what elements, attributes are valid, and how they related to each other.
3. The next is the LEMS description of the model—the dynamics. We call this the Component type declaration.

1. That’s it. We’ll see an example now.

1. The schema is defined as an XSD: XML schema document that formally defines what an XML file can look like.
2. In this example, of a simple cell model, the Izhikevich model, this lists what its attributes are.
3. In programming jargon, we're defining the structure of the class—what parameters/attributes can an instance/object of class contain.

1. The next is the LEMS description of the model—the dynamics. We call this the Component type declaration.

1. Here is the LEMS component type definition, without the dynamics for the moment.
2. What you'll notice is that a lot of it is very similar to the XSD definition.

1. I’ve written the attributes/parameters together here so you can see that there’s a one-to-one correspondence between the two model descriptions.

1. This now shows the dynamics of the component type.
2. It include information on how the states and time derivatives change, how the various variables interact.
3. This information is sufficient to then be able to create an object of this type and to simulate it.

1. Hopefully this gives you an idea of how NeuroML is modular, structured, and hierarchical by design.
2. So, we like to think of it building blocks, and since the relationships between blocks is well defined, the blocks can be easily re-used and combined to create new ones.
3. So, there are three conductances here, for example—using the same three HH type gates.
4. These conductances can be used in different cells—here two cells with different morphologies.
5. The cells can then be used in populations to create a network, and so on.

1. An important aspect of the standard is that it includes lots of commonly used model elements already.
2. So that users don’t have to write these themselves, they can use the ones already provided.
3. The mind map shows you a sub-set of model elements included in the NeuroML standard.

1. An important aspect of the standard is that it includes lots of commonly used model elements already.
2. So that users don’t have to write these themselves, they can use the ones already provided.
3. The mind map shows you a sub-set of model elements included in the NeuroML standard.

1. These tools form the core NeuroML tools—ones that the NeuroML Editors, we develop and maintain
2. These provide the basic APIs required to work with NeuroML models—to create them, to read, write, and modify them
3. PyNeuroML is the suggested tool for use—we don’t want people writing XML.

1. PyNeuroML is the suggested tool for use—we don't want people writing XML.
2. PyNeuroML includes functions to support the model life cycle
3. These can be accessed programmatically, but we also provide CLIs to make it easier for users.

1. Now, we do not want users to write XML at all
2. The Python API is generated from the schema, and one can use this to write models.

1. Once a model has been created, it is stored in its NeuroML/LEMS XML form
2. Because LEMS is formally defined and machine readable, this can now be easily converted into any other require format
3. The different simulators support NeuroML in different ways
4. For example, for NEURON, we generate NEURON scripts
5. But NetPyNE imports NeuroML to convert it into its own internal format
6. The advantage here is that simulator developers are free to choose how they want to support NeuroML

1. We've already touched on validation a little, but let's look at it in more detail
2. The first level of validation is against the schema—is the model valid, does it have the right structure
3. Since we have access to the complete model description, we can also run additional global checks—are connections correctly defined, for example?
4. Once the model has been converted to its expanded LEMS form, we run more checks
5. Does the model support the requested simulator?
6. LEMS is dimension aware—so it checks to see if units and dimensions are consistent
7. We can now simulate the model being fairly confident that it has been defined correctly
8. We can then run more tests—do we get the same results everywhere? Does it produce the right behaviour?