Scalable algorithms for Markov process parameter inference

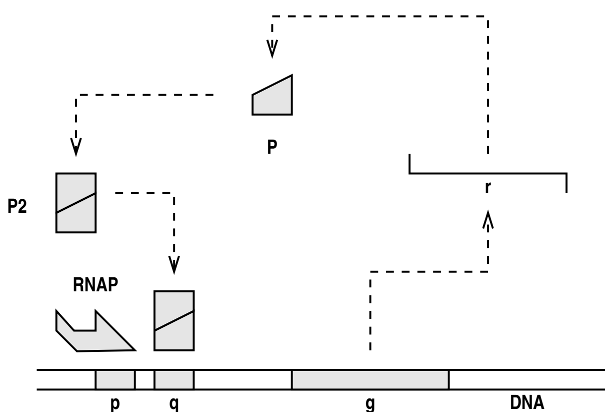
Darren Wilkinson
@darrenjw
tinyurl.com/darrenjw
School of Mathematics & Statistics,
Newcastle University, UK

Stochastic numerical algorithms
ICERM, Brown, RI, USA
18th–22nd July, 2016

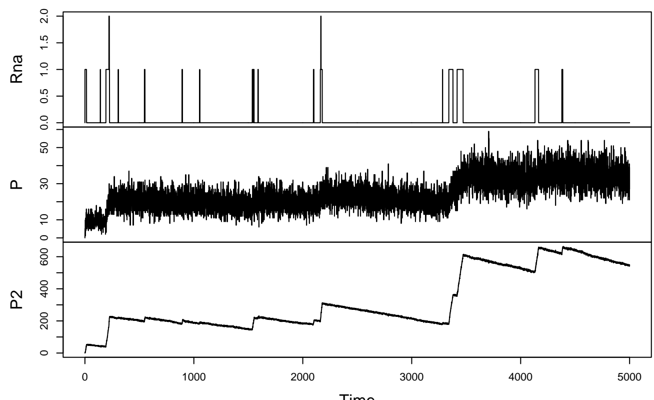
Overview

- Stochastic reaction networks, stochastic simulation and partially observed Markov process (POMP) models
- Likelihood-free (PMMH) pMCMC — exact fully Bayesian joint inference for the model state and static parameters
- (Adaptive) ABC(–SMC)
- Functional programming approaches for scalable scientific and statistical computing

Example — genetic auto-regulation



Simulated realisation of the auto-regulatory network



Multi-scale simulation issues

- Although not a huge model, this is actually rather challenging, for both simulation and inference
- Some species have very low copy number (predominantly zero or one), making the diffusion approximation unappealing, and some have fairly high copy number (several hundred)
- The high and low copy-number species are tightly coupled, leading to very discrete bursty stochastic behaviour even in the high copy-number species
- There are also two pairs of fast binding/unbinding reactions, making the problem difficult for time-stepping methods such as τ-leaping
- A rather sophisticated hybrid multi-scale algorithm would be required to simulate this process very accurately and very fast...

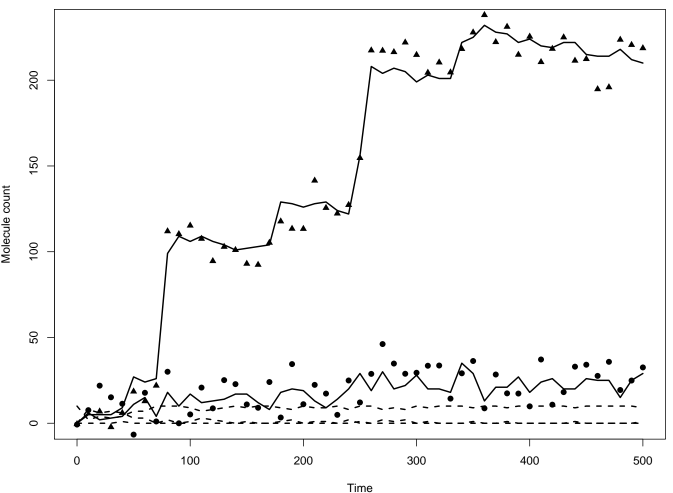
Modularity: decoupling models from simulation algorithms

- For forward modelling, there are clear and considerable benefits to separating the representation of the model from the algorithm used to simulate realisations from it:
  - There are numerous exact and approximate simulation algorithms, as well as algorithms for static model analysis — by decoupling the models from the algorithms, it is easy to apply any algorithm to any model of interest — improvements in simulation algorithms automatically apply to all models of interest
  - Modifying a model representation will typically be much easier than modifying an algorithm to simulate that model
  - When assembling large models from smaller model components, it is often more straightforward to compose model representations than associated simulation algorithms
- Few disadvantages: limit to flexibility of representation, slight inefficiencies, more sophisticated programming required

Parameter inference

- The auto-regulatory network model contains 5 species and 8 reactions
- Each reaction has an associated rate constant — these 8 rate constants may be subject to uncertainty
- The initial state of the model (5 species levels) may also be uncertain/unknown
- There could also be uncertainty about the structure of the reaction network itself — eg. presence/absence of particular reactions — this can be embedded into the parameter inference problem, but is often considered separately, and is not the subject of this talk
- We will focus here on using time course data on some aspect of one (or more) realisations of the underlying stochastic process in order to make inferences for any unknown parameters of the model

Partial, noisy data on the auto-reg model



Classes of Bayesian Monte Carlo algorithms

In this context there are 3 main classes of MC algorithms:

- ABC algorithms (likelihood–free)
    - Completely general (in principle) “global” Approximate Bayesian Computation algorithms, so just require a forward simulator, and don’t rely on (eg.) Markov property, but typically very inefficient and approximate
- POMP algorithms (likelihood–free)
    - Typically “local” (particle) MCMC-based algorithms for Partially Observed Markov Processes, again only requiring a forward simulator, but using the Markov property of the process for improved computational efficiency and “exactness”
- Likelihood-based MCMC algorithms
    - More efficient (exact) MCMC algorithms for POMP models, working directly with the model representation, not using a forward simulator, and requiring the evaluation of likelihoods associated with the sample paths of the stochastic process

Modularity and model decoupling for inference

- Decoupling the model from the inference algorithm is just as important as separation of the model from a forward simulation algorithm
- The key characteristic of likelihood-free (or "plug-and-play") algorithms is that they separate inference algorithms from the forward simulator completely — this strong decoupling has many advantages, with the main disadvantage being the relative inefficiency of the inferential algorithms
- The likelihood-free algorithms rely heavily on forward simulation, so can immediately benefit from improvements in exact and approximate simulation technology
- There is no reason why efficient likelihood-based MCMC algorithms can't also be decoupled from the model representation, but doing so for a reasonably large and flexible class of models seems to be beyond the programming skills of most statisticians...

Partially observed Markov process (POMP) models

- Continuous-time Markov process: $\mathbf{X} = \{X_s | s \geq 0\}$ (for now, we suppress dependence on parameters, θ)
- Think about integer time observations (extension to arbitrary times is trivial): for $t \in \mathbb{N}$, $\quad \mathbf{X}_t = \{X_s | t-1 < s \leq t\}$
- Sample-path likelihoods such as $\pi(\mathbf{x}_t | x_{t-1})$ can often (but not always) be computed (but are often computationally difficult), but discrete time transitions such as $\pi(x_t | x_{t-1})$ are typically intractable
- Partial observations: $\mathcal{Y} = \{y_t | t = 1, 2, \ldots, T\}$ where

$$y_t | X_t = x_t \sim \pi(y_t | x_t), \qquad t = 1, \ldots, T,$$

  where we assume that $\pi(y_t | x_t)$ can be evaluated directly (simple measurement error model)

Bayesian inference for latent process models

- Vector of model parameters, θ, the object of inference
- Prior probability distribution on θ, denoted $\pi(\theta)$
- Conditional on θ, we can simulate realisation of the stochastic process $\mathbf{X}$, with probability model $\pi(\mathbf{x}|\theta)$, which may be intractable
- Observational data $\mathcal{Y}$, determined from $\mathbf{x}$ and θ by a the probability model $\pi(\mathcal{Y}|\mathbf{x},\theta)$ — for "exact" algorithms we typically require that this model is tractable, but for ABC, we only need to be able to simulate from it
- Joint model $\pi(\theta,\mathbf{x},\mathcal{Y}) = \pi(\theta)\pi(\mathbf{x}|\theta)\pi(\mathcal{Y}|\mathbf{x},\theta)$
- Posterior distribution $\pi(\theta,\mathbf{x}|\mathcal{Y}) \propto \pi(\theta,\mathbf{x},\mathcal{Y})$
- If using Monte Carlo methods, easy to marginalise out $\mathbf{x}$ from samples from the posterior to get samples from the parameter posterior $\pi(\theta|\mathcal{Y})$

Likelihood-free PMMH pMCMC

- Particle Markov chain Monte Carlo (pMCMC) methods are a powerful tool for parameter inference in POMP models
- In the variant known as particle marginal Metropolis Hastings (PMMH), a (random walk) MH MCMC algorithm is used to explore parameter space, but at each iteration, a (bootstrap) particle filter (SMC algorithm) is run to calculate terms required in the acceptance probability
- The "magic" of pMCMC is that despite the fact that the particle filters are "approximate", pMCMC algorithms nevertheless have the "exact" posterior distribution of interest (either $\pi(\theta|\mathcal{Y})$ or $\pi(\theta, \mathbf{x}|\mathcal{Y})$) as their target
- If a sophisticated particle filter is used, pMCMC can be a reasonably efficient likelihood-based MCMC method — however, when a simple "bootstrap" particle filter is used, the entire process is "likelihood-free", but still "exact"

Particle MCMC (pMCMC)

- pMCMC is the only obvious practical option for constructing a global likelihood-free MCMC algorithm for POMP models which is exact (Andrieu et al, 2010)
- Start by considering a basic marginal MH MCMC scheme with target $\pi(\theta|\mathcal{Y})$ and proposal $f(\theta^\star|\theta)$ — the acceptance probability is $\min\{1, A\}$ where

$$A = \frac{\pi(\theta^\star)}{\pi(\theta)} \times \frac{f(\theta|\theta^\star)}{f(\theta^\star|\theta)} \times \frac{\pi(\mathcal{Y}|\theta^\star)}{\pi(\mathcal{Y}|\theta)}$$

- We can't evaluate the final terms, but if we had a way to construct a Monte Carlo estimate of the likelihood, $\hat{\pi}(\mathcal{Y}|\theta)$, we could just plug this in and hope for the best:

$$A = \frac{\pi(\theta^\star)}{\pi(\theta)} \times \frac{f(\theta|\theta^\star)}{f(\theta^\star|\theta)} \times \frac{\hat{\pi}(\mathcal{Y}|\theta^\star)}{\hat{\pi}(\mathcal{Y}|\theta)}$$

"Exact approximate" MCMC (the pseudo-marginal approach)

- Remarkably, provided only that $\mathrm{E}[\hat{\pi}(\mathcal{Y}|\theta)] = \pi(\mathcal{Y}|\theta)$, the stationary distribution of the Markov chain will be exactly correct (Beaumont, 2003, Andrieu & Roberts, 2009)
- Putting $W = \hat{\pi}(\mathcal{Y}|\theta)/\pi(\mathcal{Y}|\theta)$ and augmenting the state space of the chain to include W, we find that the target of the chain must be

$$\propto \pi(\theta)\hat{\pi}(\mathcal{Y}|\theta)\pi(w|\theta) \propto \pi(\theta|\mathcal{Y})w\pi(w|\theta)$$

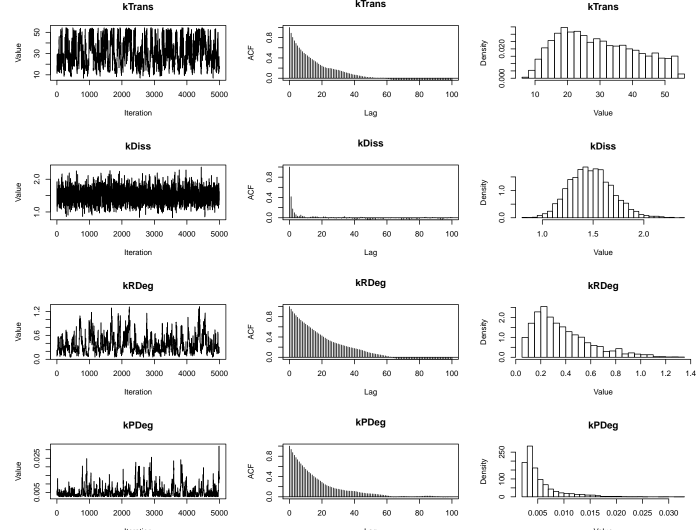
  and so then the above "unbiasedness" property implies that $\mathrm{E}(W|\theta) = 1$, which guarantees that the marginal for θ is exactly $\pi(\theta|\mathcal{Y})$

Particle marginal Metropolis-Hastings (PMMH)

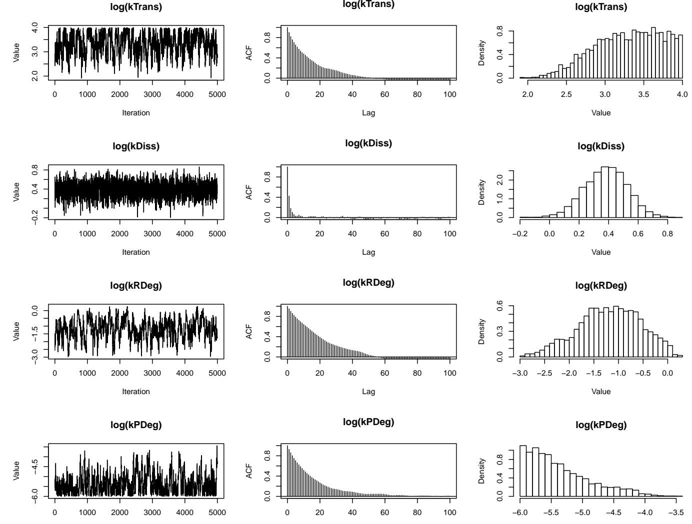
- Likelihood estimates constructed via importance sampling typically have this "unbiasedness" property, as do estimates constructed using a particle filter
- If a particle filter is used to construct the Monte Carlo estimate of likelihood to plug in to the acceptance probability, we get (a simple version of) the particle Marginal Metropolis Hastings (PMMH) pMCMC algorithm
- The full PMMH algorithm also uses the particle filter to construct a proposal for $\mathbf{x}$, and has target $\pi(\theta, \mathbf{x}|\mathcal{Y})$ — not just $\pi(\theta|\mathcal{Y})$
- The (bootstrap) particle filter relies only on the ability to forward simulate from the process, and hence the entire procedure is "likelihood-free"

See Golightly and W (2011) for further details

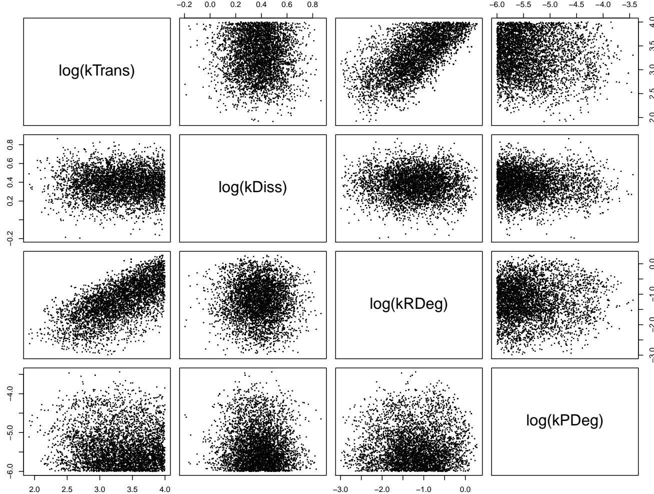
PMMH inference results



PMMH inference results



PMMH inference results



“Sticking” and tuning of PMMH

- As well as tuning the θ proposal variance, it is necessary to tune the number of particles, N in the particle filter — need enough to prevent the chain from sticking, but computational cost roughly linear in N
- Number of particles necessary depends on θ, but don't know θ *a priori*
- Initialising the sampler is non-trivial, since much of parameter space is likely to lead to likelihood estimates which are dominated by noise — how to move around when you don't know which way is “up”?!
- Without careful tuning and initialisation, burn-in, convergence and mixing can all be very problematic, making algorithms painfully slow...

Alternative: approximate Bayesian computation (ABC)

- Since $\pi(\theta, \mathbf{x}, \mathcal{Y}) = \pi(\theta)\pi(\mathbf{x}|\theta)\pi(\mathcal{Y}|\theta, \mathbf{x})$, it is trivial to generate samples from $\pi(\theta, \mathbf{x}, \mathcal{Y})$ and to marginalise these down to $\pi(\theta, \mathcal{Y})$
- Exact rejection algorithm: generate $(\theta^\star, \mathcal{Y}^\star)$ from $\pi(\theta, \mathcal{Y})$ and keep provided that $\mathcal{Y} = \mathcal{Y}^\star$ otherwise reject and try again
- This gives exact realisations from $\pi(\theta|\mathcal{Y})$, but in practice the acceptance rate will be very small (or zero)
- ABC: Define a metric on the sample space, $\rho(\cdot, \cdot)$, and accept $(\theta^\star, \mathcal{Y}^\star)$ if $\rho(\mathcal{Y}, \mathcal{Y}^\star) < \varepsilon$
- This gives exact realisations from $\pi(\theta|\rho(\mathcal{Y}, \mathcal{Y}^\star) < \varepsilon)$, which tends to the true posterior as $\varepsilon \longrightarrow 0$
- Still problematic if there is a large discrepancy between the prior and posterior...

Summary statistics

- The choice of metric $\rho(\cdot,\cdot)$ is very important to the overall efficiency and performance of ABC methods
- Using a naive Euclidean distance on the raw data $\rho(\mathcal{Y},\mathcal{Y}^\star)=\|\mathcal{Y}-\mathcal{Y}^\star\|$ is likely to perform poorly in practice — even with a perfect choice of parameters, it is extremely unlikely that you will "hit" the data
- Ideally, we would use a vector of sufficient statistics, $s(\mathcal{Y})$ of the likelihood model associated with the process, to summarise the important aspects of the data relevant to parameter inference, and then define a metric on $s(\cdot)$
- In practice, for complex models we don't know the sufficient statistics (and they probably don't exist), but we nevertheless form a vector of summary statistics, which we hope capture the important aspects of the process and ignore the irrelevant noise in each realisation

Summary statistics for POMP models

- For time series data, obvious summary statistics for each univariate component include the sample mean, (log-)variance, and the lag 1 and 2 auto-correlations
- For multivariate time series, the (matrix of) cross-correlation(s) is also useful
- Together these summary statistics capture much of the essential dynamic structure of multivariate time series
- For our auto-reg network example, this reduces the dimension of the statistic we are trying to match from $50 \times 2 = 100$ to $2 \times 4 + 1 = 9$ — not only a big reduction, but we are now trying to match statistics that are relatively robust to noise in the data
- The individual components of $s(\cdot)$ are on different scales, so need to re-weight (eg. by standardising) before applying a simple metric such as Euclidean distance

Issues with simple rejection ABC

- There are two main problems with naive rejection sampling based ABC:
    - The first relates to the dimension of the data, and this is (largely) dealt with by carefully choosing and weighting appropriate summary statistics
    - The second relates to the dimension of the parameter space...
- If the dimension of the parameter space is large, the posterior distribution is likely to have almost all of its mass concentrated in a tiny part of the space covered by the prior, so the chances of hitting on good parameters when sampling from the prior will be very small
- Might be better to gradually "zoom in" on promising parts of the parameter space gradually over a series of iterations...

ABC–SMC

- Interest in a Bayesian posterior distribution

$$\pi(\theta|x) \propto \pi(\theta)f(x|\theta)$$

  where $f(x|\theta)$ is intractable
- Observed data x_0
- Sequence of approximations

$$\pi_t(\theta) = \pi(\theta|\rho(x, x_0) < \varepsilon_t),$$

  where $\infty = \varepsilon_0 > \varepsilon_1 > \cdots > \varepsilon_n > 0$ and $\rho(\cdot, \cdot)$ is a suitable metric on data space
- π_0 is the prior, and for sufficiently small ε_n, hopefully π_n not too far from the posterior, $\pi(\theta|x_0)$
- Progressively reduce tolerances to improve agreement between successive distributions and hopefully improve acceptance rates

ABC–SMC algorithm

- Suppose we have a large (possibly weighted) sample of size N from $\pi_{t-1}(\theta)$, $\{\theta_1, \ldots, \theta_N\}$ with normalised weights $\tilde{w}_{t-1}^i$
- Pick $\theta_i^\star \sim \pi_{t-1}(\theta)$ (weighted particles)
- Perturb $\theta_i^{\star\star} \sim K(\theta_i^\star, \theta_i^{\star\star})$
- Simulate $x^\star \sim f(x^\star|\theta^{\star\star})$ from intractable likelihood model
- Accept only if $\rho(x^\star, x_0) < \varepsilon_t$, otherwise reject go back to the start and pick another $\theta_i^\star$
- Keep $\theta^{\star\star}$ as ith member of new sample, and compute its weight as

$$w_t^i = \frac{\pi(\theta^{\star\star})}{\sum_{j=1}^N \tilde{w}_{t-1}^i K(\theta_j, \theta^{\star\star})}$$

- Once a sample of size N is obtained, normalise weights and increment t

Pros and cons of ABC(–SMC)

- All likelihood-free methods have a tendency to be very computationally intensive and somewhat inefficient
- ABC is very general, and can be applied to arbitrary settings (eg. not just POMP models)
- ABC methods parallelise very well, and hence can be useful for getting reasonably good approximations to be true posterior relatively quickly if suitable hardware is available
- ABC is becoming increasingly popular outside of statistics, where the idea of "moment matching" is familiar and intuitive
- ABC usually results in a distribution significantly over-dispersed relative to the true posterior
- The tuning parameters can affect the ABC posterior
- It's hard to know how well you are doing when working "blind"

Pros and cons of pMCMC

- Most obvious application is to POMP models — less general than ABC
- It targets the "exact" posterior distribution, irrespective of the choices of tuning constants!
- In practice, for finite length runs, the pMCMC output tends to be slightly under-dispersed relative to the true posterior ("missing the tails")
- Parallelises fine over multiple cores on a single machine, but less well over a cluster
- Although the theory underpinning pMCMC is non-trivial, implementing likelihood-free PMMH is straightforward, and has the advantage that it targets the "exact" posterior distribution

ABC–SMC and PMMH

- PMMH algorithms require careful tuning, and are not trivial to parallelise
- Multiple parallel chains benefit from being initialised at samples from the posterior, in order to minimise burn-in
- Tuning the number of particles to use in the particle filter is best done by averaging over a sample from the posterior
- ABC–SMC can be used to generate a sample from an approximation to the posterior, which is good enough for tuning and initialisation of PMMH chains
  - ABC algorithms parallelise well, so this strategy is well suited to taking advantage of parallel hardware
  - ABC–SMC algorithms also require tuning, but reasonable default choices work mostly OK for POMP models
- Multiple independent tuned PMMH chains can then target the exact posterior

Summary

- For conducting Bayesian inference for complex simulation models, "likelihood–free" methods are very attractive
- There are many likelihood–free algorithms, some of which are "exact" — pMCMC algorithms being a notable example
- Likelihood-free algorithms can sometimes be very inefficient
- pMCMC not the only option worth considering — ABC–SMC methods, and SMC^2 are also worth trying; also iterated filtering for a ML solution
- Hybrid procedures which parallelise well and take advantage of the particular strengths of different algorithms are worth exploring

Problems with parallelising code

- Most problems arise from concurrent processes needing to access and modify some data — shared mutable state
  - Synchronisation (threads standing idle waiting for other tasks to complete)
  - Deadlocks (two or more competing actions are each waiting for the other to finish, and thus neither ever does)
  - Race conditions (unpredictable behaviour depending on timing of unpredictable events - eg. simultaneous updating of data or a resource)
- A major issue is that we are using programming languages and models of computation from the dawn of the computing age — think about how much computing hardware has changed in the last 40 years compared to the languages that most people use for scientific computing...

Functional approaches to concurrency and parallelisation

- Functional languages emphasise immutable state and referentially transparent functions
- Immutable state, and referentially transparent (side-effect free) declarative workflow patterns are widely used for systems which really need to scale (leads to naturally parallel code)
- Shared mutable state is the enemy of concurrency and parallelism (synchronisation, locks, waits, deadlocks, race conditions, ...) — by avoiding mutable state, code becomes easy to parallelise
- The recent resurgence of functional programming and functional programming languages is partly driven by the realisation that functional programming provides a natural way to develop algorithms which can exploit multi-core parallel and distributed architectures, and efficiently scale

Category theory

Dummies guide:

- A “collection” (or parametrised “container” type) together with a “map” function (defined in a sensible way) represents a functor
- If the collection additionally supports a (sensible) “apply” operation, it is an applicative
- If the collection additionally supports a (sensible) “flattening” operation, it is a monad (required for composition)
- For a “reduce” operation on a collection to parallelise cleanly, the type of the collection together with the reduction operation must define a monoid (must be an associative operation, so that reductions can proceed in multiple threads in parallel)

Data structures and parallelism

- Scala has an extensive "Collections framework" (http://docs.scala-lang.org/overviews/collections/overview.html), providing a large range of data structures for almost any task (Array, List, Vector, Map, Range, Stream, Queue, Stack, ...)
- Most collections available as either a mutable or immutable version — idiomatic Scala code favours immutable collections
- Most collections have an associated parallel version, providing concurrency and parallelism "for free" (examples later)
- As the (immutable) collections are monads, they are often referred to as monadic collections

Spark

- Spark is a scalable analytics library, including some ML (from Berkeley AMP Lab)
- It is rapidly replacing Hadoop and MapReduce as the standard library for big data processing and analytics
- It is written in Scala, but also has APIs for Java and Python (and experimental API for R)
- Only the Scala API leads to both concise elegant code and good performance
- Central to the approach is the notion of a resilient distributed dataset (RDD) — a monadic collection — hooked up to Spark via a connector

Lazy, functional, immutable data workflows are what makes it all work — that's why it's implemented in a language like Scala

Spark example

- Example of a logistic regression log-likelihood gradient computation taken from Spark MLLib:

```
val gradient = points.map { p =>
  p.x * (1/(1+exp(-p.y*(w.dot(p.x))))-1) *
  p.y}.reduce(_ + _)
```

- Note that the RDD `points` could be terabytes big and distributed across a large cluster of nodes, yet the code is written as for any other immutable collection
- The code itself is agnostic about the type of the collection and whether it is parallel or distributed — this is a separate concern
- The `map` and `reduce` operations naturally parallelise

Spark combinators

- Variety of lazy, functional combinators for RDDs which can be chained together and optimised prior to execution
  - Transformations: map, filter, flatMap, sample, union, intersection, groupByKey, ...
  - Actions: reduce, collect, count, take, takeSample, foreach, ...
- Higher-level tools include Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming
- Intelligent scheduling of operations to maximise data locality and minimise communication overhead
- Spark RDDs can be used to distribute computation as well as data — not only useful for "big data" problems — good for "big models" too...

Example: Scalable particle filter

First define a typeclass describing the required properties of collections, independently of whether the actual concrete implementation to be used is serial, parallel or distributed

```
trait GenericColl[C[_]] {
  def map[A, B](ca: C[A])(f: A => B): C[B]
  def reduce[A](ca: C[A])(f: (A, A) => A): A
  def flatMap[A, B, D[B] <: GenTraversable[B]]
                              (ca: C[A])(f: A => D[B]): C[B]
  def zip[A, B](ca: C[A])(cb: C[B]): C[(A, B)]
  def length[A](ca: C[A]): Int
}
```

Example: Scalable particle filter

Code for one step of a particle filter, for any generic collection of particles

```
def update[S: State, O: Observation, C[_]: GenericColl](
  dataLik: (S, O) => LogLik, stepFun: S => S
)(x: C[S], o: O): (LogLik, C[S]) = {
  val xp = x map (stepFun(_))
  val lw = xp map (dataLik(_, o))
  val max = lw reduce (math.max(_, _))
  val rw = lw map (lwi => math.exp(lwi - max))
  val srw = rw reduce (_ + _)
  val l = rw.length
  val z = rw zip xp
  val rx = z flatMap (p => Vector.fill(
                      Poisson(p._1 * l / srw).draw)(p._2))
  (max + math.log(srw / l), rx)
}
```

Example: Scalable particle filter

A particle filter is then just a fold over a sequence of observations

```
def pFilter[S: State, O: Observation, C[_]: GenericColl,
                                        D[O] <: GenTraversable[O]](
  x0: C[S], data: D[O], dataLik: (S, O) => LogLik,
      stepFun: S => S
  ): (LogLik, C[S]) = {
    val updater = update[S, O, C](dataLik, stepFun) _
    data.foldLeft((0.0, x0))((prev, o) => {
      val next = updater(prev._2, o)
      (prev._1 + next._1, next._2)
    })
  }

def pfMll[S: State, P: Parameter, O: Observation,
              C[_]: GenericColl, D[O] <: GenTraversable[O]](
          simX0: P => C[S], stepFun: P => S => S,
          dataLik: P => (S, O) => LogLik, data: D[O]
      ): (P => LogLik) = (th: P) => pFilter(simX0(th),
                          data, dataLik(th), stepFun(th))._1
```

Summary

- Convincing arguments can be made that strongly typed functional programming languages scale better than conventional imperative (O–O) languages
- Concurrency and parallelism are difficult to manage in imperative languages due to shared mutable state
- Since functional languages avoid mutable state, writing concurrent and parallel code in functional languages is simple, natural and elegant
- Understanding the role of immutability and referential transparency in functional programming will change how you think about computation itself
- Category theory provides the "design patterns" for functional programming

References

Golightly, A. and D. J. Wilkinson (2011) Bayesian parameter inference for stochastic biochemical network models using particle MCMC. *Interface Focus*, **1**(6):807–820.

Golightly, A. and Wilkinson, D. J. (2015) Bayesian inference for Markov jump processes with informative observations, *Statistical Applications in Genetics and Molecular Biology*, **14**(2):169–188.

Owen, J., Wilkinson, D. J., Gillespie, C. S. (2015) Scalable inference for Markov processes with intractable likelihoods, *Statistics and Computing*, **25**(1):145–156.

Owen, J., Wilkinson, D. J., Gillespie, C. S. (2015) Likelihood free inference for Markov processes: a comparison, *Statistical Applications in Genetics and Molecular Biology*, **14**(2):189-209.

Wilkinson, D. J. (2011) *Stochastic Modelling for Systems Biology, second edition*. Chapman & Hall/CRC Press.

@darrenjw darrenjw.wordpress.com