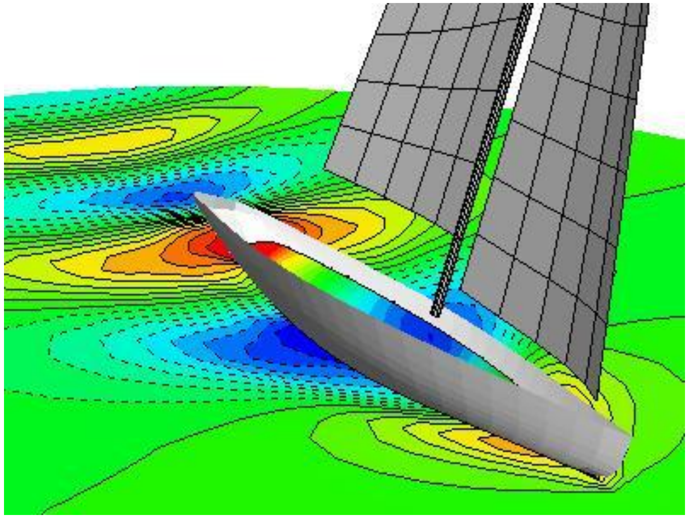
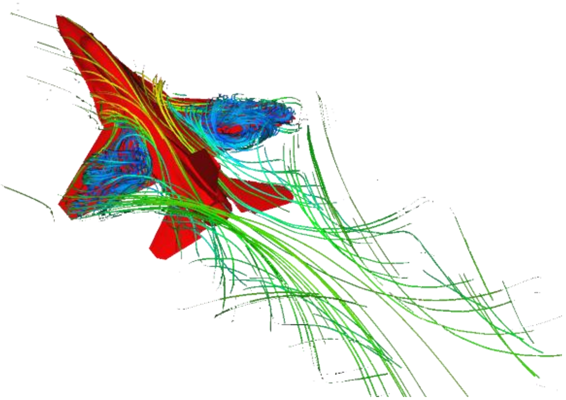
CS 179: GPU Computing

Lecture 16: Simulations and Randomness

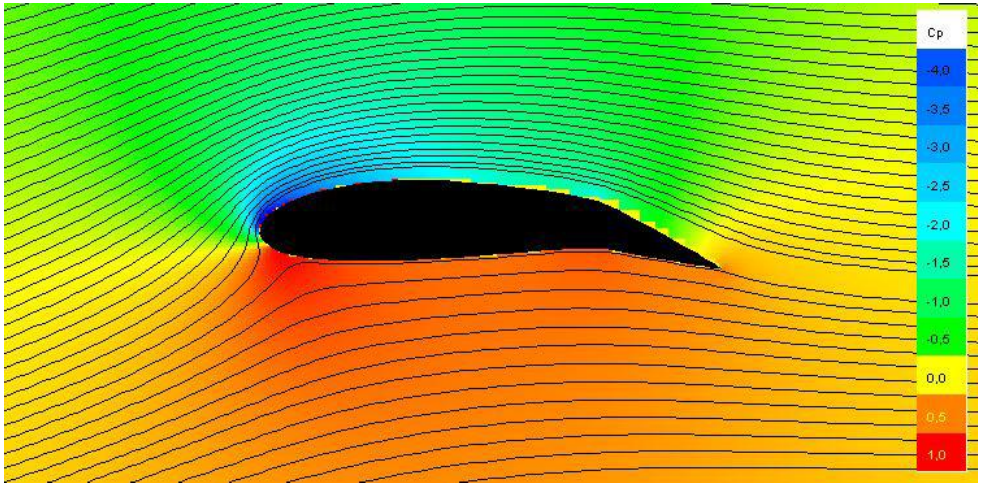
Simulations



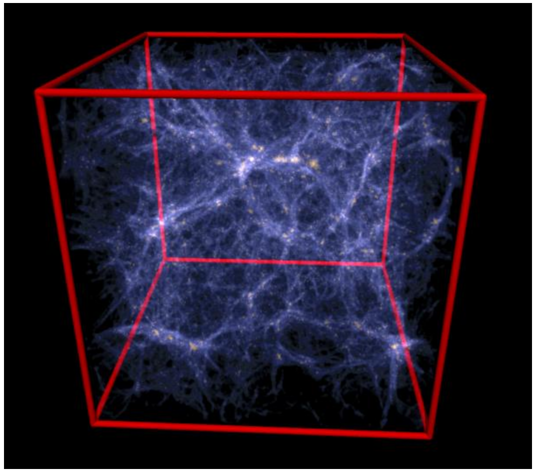
South Bay Simulations,
http://www.panix.com/~brosen/graphics/iacc.400.jpg



Exa Corporation, http://www.exa.com/images/f16.png



Flysurfer Kiteboarding, http://www.flysurfer.com/wp-content/blogs.dir/3/files/gallery/research-and-development/zwischenablage07.jpg



Max-Planck Institut, http://www.mpa-garching.mpg.de/gadget/hydrosims/

Simulations

- But what if your problem is hard to solve? e.g.
  - EM radiation attenuation
  - Estimating complex probability distributions
  - Complicated ODEs, PDEs
    - (e.g. option pricing in last lecture)
  - Geometric problems w/o closed-form solutions
    - Volume of complicated shapes

Simulations

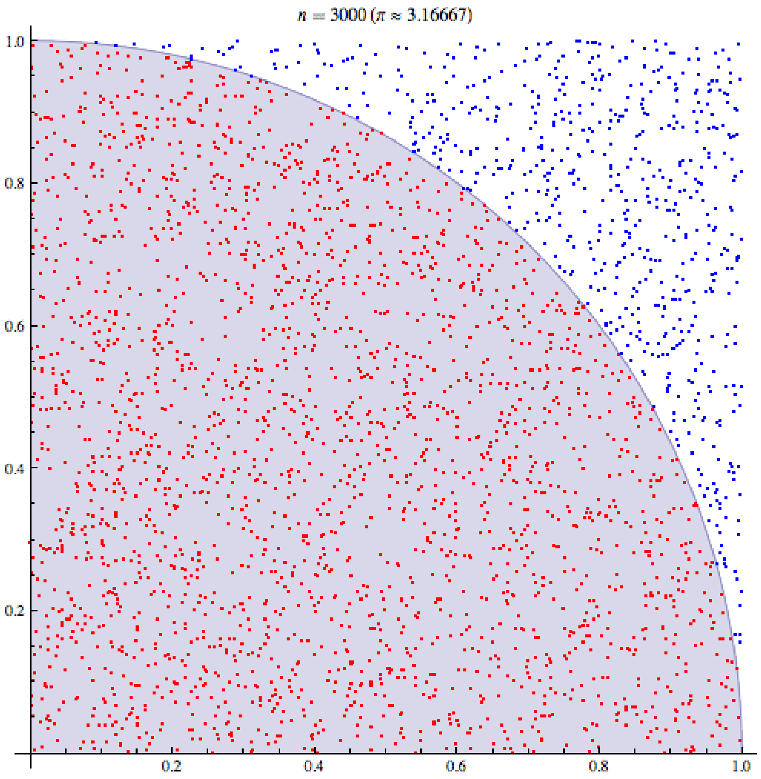
- Potential solution: **Monte Carlo methods**
  - Run simulation with randomly chosen inputs
    - (Possibly according to some distribution)
  - Do it again… and again… and again…
  - Aggregate results

Monte Carlo example

- Estimating the value of π

Monte Carlo example

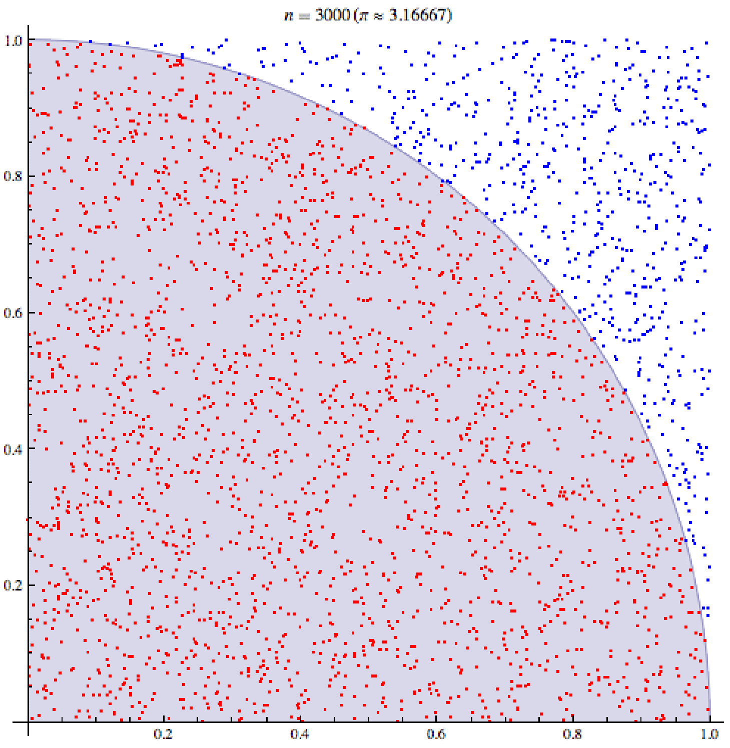
- Estimating the value of π
  - Quarter-circle of radius r:
    - Area = $(\pi r^2)/4$
  - Enclosing square:
    - Area = r^2
  - Fraction of area: π/4



"Pi 30K" by CaitlinJo - Own workThis mathematical image was created with Mathematica. Licensed under CC BY 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Pi_30K.gif#/media/File:Pi_30K.gif

Monte Carlo example

- Estimating the value of π
  - Quarter-circle of radius r:
    - Area = $(\pi r^2)/4$
  - Enclosing square:
    - Area = r^2
  - Fraction of area: $\pi/4 \approx 0.79$



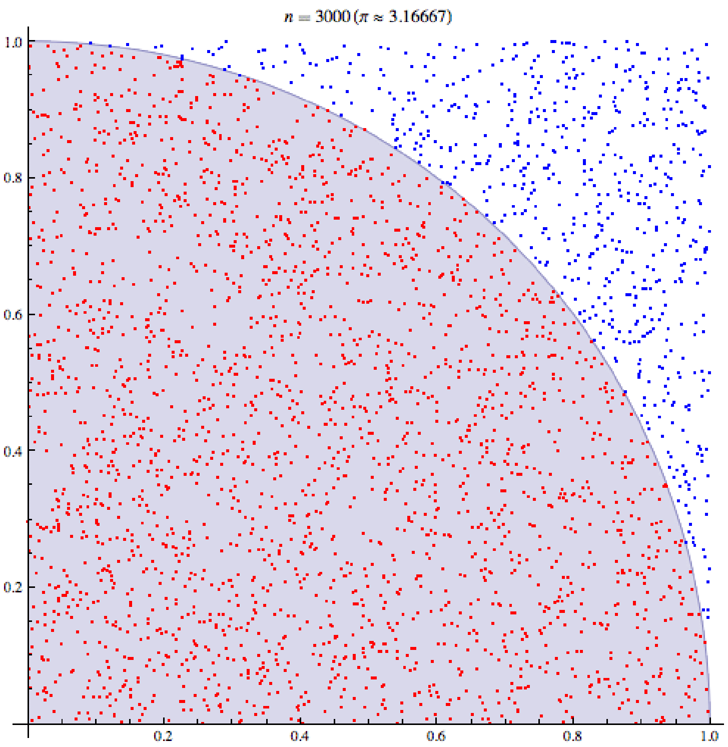
- "Solution": Randomly generate lots of points, calculate fraction within circle
  - Answer should be pretty close!

"Pi 30K" by CaitlinJo - Own workThis mathematical image was created with Mathematica. Licensed under CC BY 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Pi_30K.gif#/media/File:Pi_30K.gif

Monte Carlo example

- Pseudocode:

```
(simulate on N points)
(assume r = 1)

points_in_circle = 0
for i = 0,…,N-1:
     randomly pick point (x,y) from
               uniform distribution in $[0,1]^2$
     if (x,y) is in circle:
          points_in_circle++

return (points_in_circle / N) * 4
```

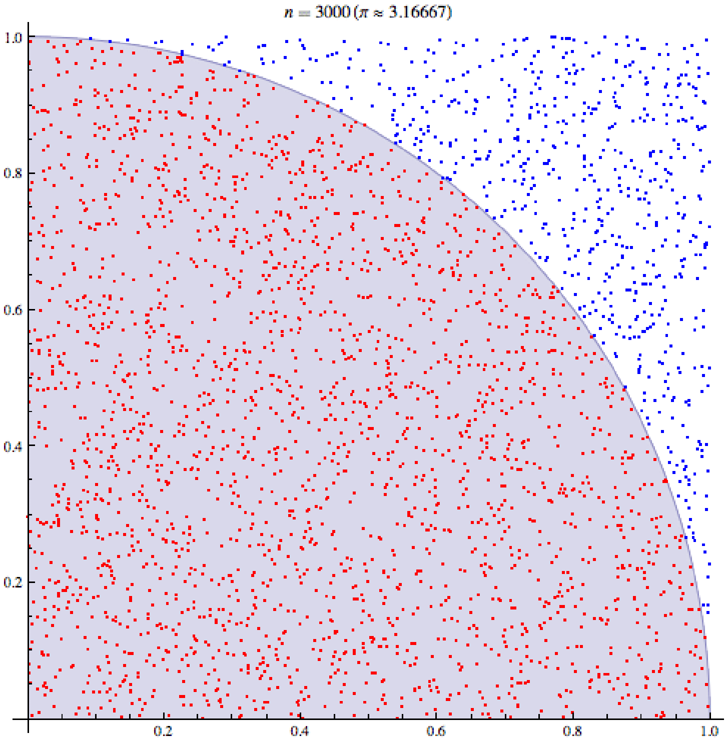


"Pi 30K" by CaitlinJo - Own workThis mathematical image was created with Mathematica. Licensed under CC BY 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Pi_30K.gif#/media/File:Pi_30K.gif

Monte Carlo example

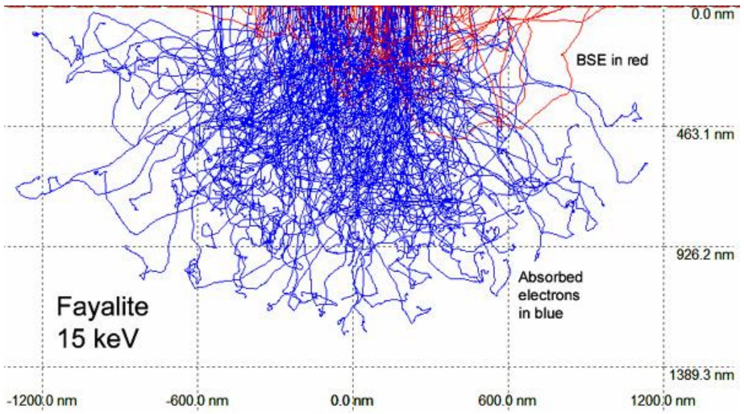
- Pseudocode:

```
(simulate on N points)
(assume r = 1)

points_in_circle = 0
for i = 0,…,N-1:
     randomly pick point (x,y) from
               uniform distribution in $[0,1]^2$
     if x^2 + y^2 < 1:
          points_in_circle++

return (points_in_circle / N) * 4
```

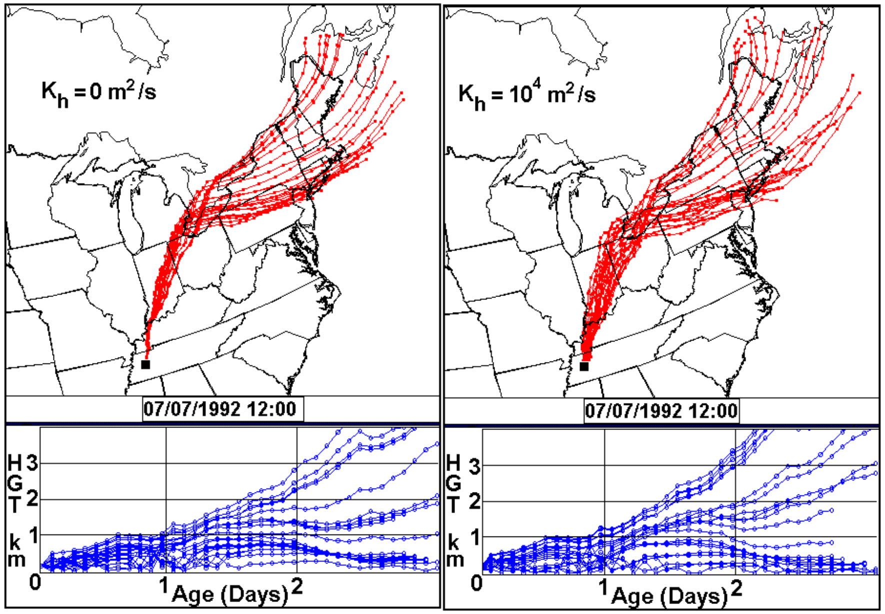


"Pi 30K" by CaitlinJo - Own workThis mathematical image was created with Mathematica. Licensed under CC BY 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Pi_30K.gif#/media/File:Pi_30K.gif

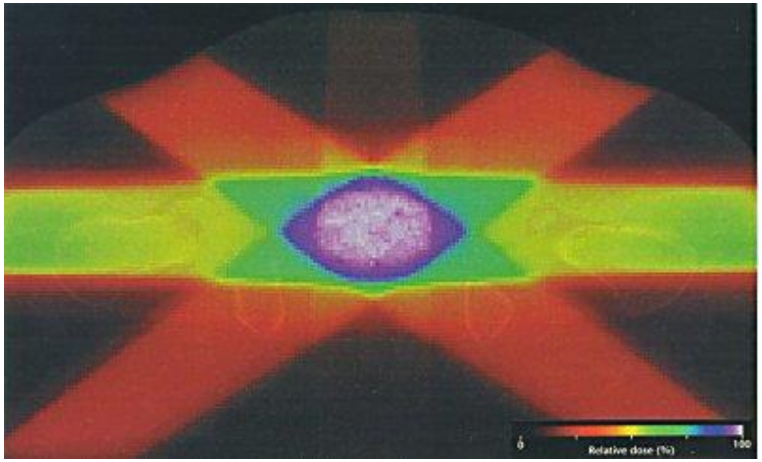
Monte Carlo simulations



Planetary Materials Microanalysis Facility, , Northern Arizona University, http://www4.nau.edu/microanalysis/microprobe-sem/Images/Monte_Carlo.jpg



Center for Air Pollution Impact & Trend Analysis, Washington University in St. Louis, http://www4.nau.edu/microanalysis/microprobe-sem/Images/Monte_Carlo.jpg



http://www.cancernetwork.com/sites/default/files/cn_import/n0011bf1.jpg

General Monte Carlo method

- Pseudocode:

```
for (number of trials):
     randomly pick value from a probability distribution
     perform deterministic computation on inputs

(aggregate results)
```

General Monte Carlo method

- Why it works:
  - Law of large numbers!

$$\overline{X}_n \to \mu \qquad \text{for} \qquad n \to \infty,$$

General Monte Carlo method

- Pseudocode:

```
for (number of trials):
     randomly pick value from a probability distribution
     perform deterministic computation on inputs

(aggregate results)
```

- Can we parallelize this?

General Monte Carlo method

- Pseudocode:

```
for (number of trials):
     randomly pick value from a probability distribution
     perform deterministic computation on inputs

(aggregate results)
```

Trials are independent

- Can we parallelize this?

General Monte Carlo method

- Pseudocode:

```
for (number of trials):
     randomly pick value from a probability distribution
     perform deterministic computation on inputs

(aggregate results)
```

Trials are independent

Usually so
(e.g. with reduction)

- Can we parallelize this?

General Monte Carlo method

- Pseudocode:

```
for (number of trials):
     randomly pick value from a probability distribution
     perform deterministic computation on inputs

(aggregate results)
```

What about this?

Trials are independent

Usually so
(e.g. with reduction)

- Can we parallelize this?

Parallelized Random Number Generation

Early Credits

- Algorithm and presentation based on:
  - “Parallel Random Numbers: As Easy as 1, 2, 3”
    - (Salmon, Moraes, Dror, Shaw) at D. E. Shaw Research
    - Developed for biomolecular simulations on Anton (massively parallel ASIC-based supercomputer)
    - Also applicable to CPUs, GPUs

Random Number Generation

- Generating random data computationally is hard
  - Computers are deterministic!

https://cdn.tutsplus.com/vector/uploads/legacy/tuts/165_Shiny_Dice/27.jpg

Random Number Generation

- Two methods:
  - Hardware random number generator
    - aka TRNG (“True” RNG)
    - Uses data collected from environment (thermal, optical, etc)
    - Very slow!
  - Pseudorandom number generator (PRNG)
    - Algorithm that produces “random-looking” numbers
    - Faster – limited by computational power

Demonstration

Random Number Generation

- PRNG algorithm should be:
  - High-quality
    - Produce “good” random data
  - Fast
    - (In its own right)
  - Parallelizable!

- Can we do it?
  - (Assume selection from uniform distribution)

A Very Basic PRNG

- “Linear congruential generator” (LCG)
  - e.g. C’s rand()

```
//from glibc

int32_t val = state[0];
val = ((state[0] * 1103515245) + 12345)
          & 0x7fffffff;
state[0] = val;
*result = val;
```

  - General formula:

$$X_{n+1} = (aX_n + c) \bmod m$$

    - X_0 is the “seed” (e.g. system time)

A Very Basic PRNG

- “Linear congruential generator” (LCG)
  - e.g. C’s rand()

```
//from glibc

int32_t val = state[0];
val = ((state[0] * 1103515245) + 12345)
          & 0x7fffffff;
state[0] = val;
*result = val;
```

  - General formula:

$$X_{n+1} = (aX_n + c) \mod m$$

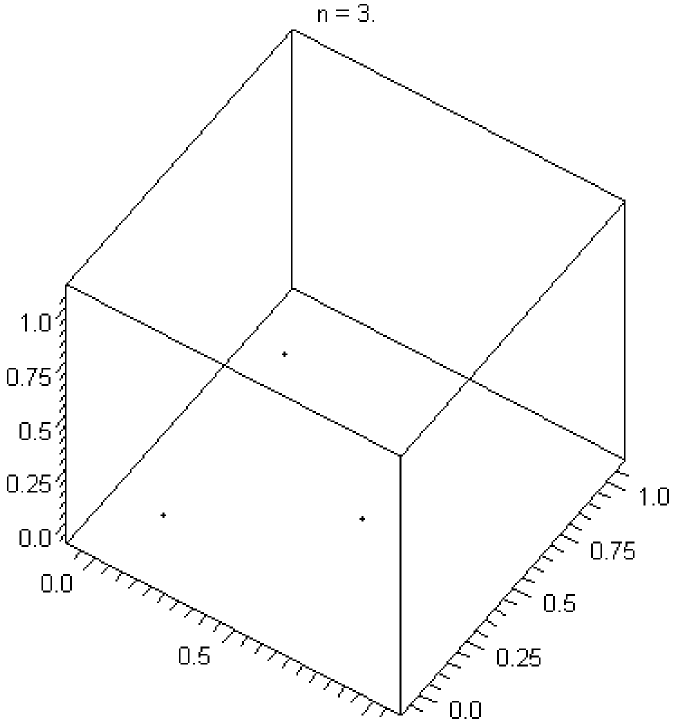
Non-parallelizable
recurrence relation!

Linear congruential generators

$$X_{n+1} = (aX_n + c) \mod m$$

- Not high quality!
  - Clearly non-uniform

- Fast to compute
- Not parallelizable!



"Lcg 3d". Licensed under CC BY-SA 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Lcg_3d.gif#/media/File:Lcg_3d.gif

Measures of RNG quality

- Impossible to prove a sequence is “random”

- Possible tests:
  - Frequency
  - Periodicity - do the values repeat too early?
  - Linear dependence
  - …

PRNG Parallelizability

- Many PRNGs (like the LCG) have a non-parallelizable appearance:

$$X_{n+1} = f(X_n)$$

  - (Better chance of good data when):
    - All X_i in some large state space
    - Complicated function f

PRNG Parallelizability

- Possible “approach” to GPU parallelization:
  - Assign a PRNG to each thread!
    - Initialize with e.g. different X_0
    - Thread 0 produces sequence $X_{n+1,0} = f(X_{n,0})$
    - Thread 1 produces sequence $X_{n+1,1} = f(X_{n,1})$
    - …

PRNG Parallelizability

- Possible “approach” to GPU parallelization:
  - Assign a PRNG to each thread!
    - Initialize with e.g. different X_0
    - Thread 0 produces sequence $X_{n+1,0} = f(X_{n,0})$
    - Thread 1 produces sequence $X_{n+1,1} = f(X_{n,1})$
    - …
  - In practice, often cannot get high quality
    - Repeated values, lack of good, enumerable parameters

PRNG Parallelizability

- Instead of:

$$X_{n+1} = f(X_n)$$

- Suppose we had:

$$X_{n+1} = b(n)$$

  - This is parallelizable! (Without our previous “trick”)

- Is this possible?

More General PRNG

- "Keyed" PRNG given by:
  - Transition function: $f\colon S \to S$
  - Output function: $g\colon K \times S \to U$

    - S: Internal (hidden) state space
    - U: Output space
    - K: "Key space"
      - Can "seed" output behavior without relying on X_0 alone – useful for scientific reproducibility!

More General PRNG

- “Keyed” PRNG given by:
  - Transition function: $f: S \rightarrow S$
  - Output function: $g: K \times S \rightarrow U$
    - S: Internal (hidden) state space
    - U: Output space
    - K: “Key space”
      - Can “seed” output behavior without relying on X_0 alone – useful for scientific reproducibility!

If S has *J* times more bits than U, can produce *J* outputs per transition.

Assume *J* = 1 in this lecture

More General PRNG

- "Keyed" PRNG given by:
  - Transition function: $f\colon S \to S$
  - Output function: $g\colon K \times S \to U$
  - "Trivial" example: LCG $X_{n+1} = (aX_n + c) \mod m$
    - $f(X_n) = aX_n + c$
    - $g(X_n) = X_n$
    - *S* is (for example) the space of 32-bit integers
    - *U = S*
    - *K* is "trivial" (no keys used)

More General PRNG

- “Keyed” PRNG given by:
  - Transition function: $f\colon S \to S$
  - Output function: $g\colon K \times S \to U$
  - “Trivial” example: LCG $X_{n+1} = (aX_n + c) \mod m$
    - $f(X_n) = aX_n + c$
    - $g(X_n) = X_n$
    - f is more complicated than g!

More General PRNG

- “Keyed” PRNG given by:
  - Transition function: $f\colon S \to S$
  - Output function: $g\colon K \times S \to U$
  - General theme: *f* is complicated, *g* is simple
    - What if we flipped that?

More General PRNG

- “Keyed” PRNG given by:
  - Transition function: $f\colon S \to S$
  - Output function: $g\colon K \times S \to U$
  - General theme: *f* is complicated, *g* is simple
    - What if we flipped that?
    - What if *f* were so simple that it could be evaluated *explicity*?

More General PRNG

- i.e. what if we had:
  - Simple transition function (p-bit integer state space):
    $$f(s) = (s+1) \bmod 2^p$$
    - This is just a counter! Can expand into explicit formula
      $$f(n) = (n + n_0) \bmod 2^p$$
    - These form **counter-based PRNGs**
  - Complicated output function g
- Would this work?

More General PRNG

- i.e. what if we had:
  - Simple transition function *f*
  - Complicated output function *g*(*k*, *n*)
    - Should be *bijective* w/r/to n
      - Guarantees period of 2^p
    - Shouldn’t be *too* difficult to compute

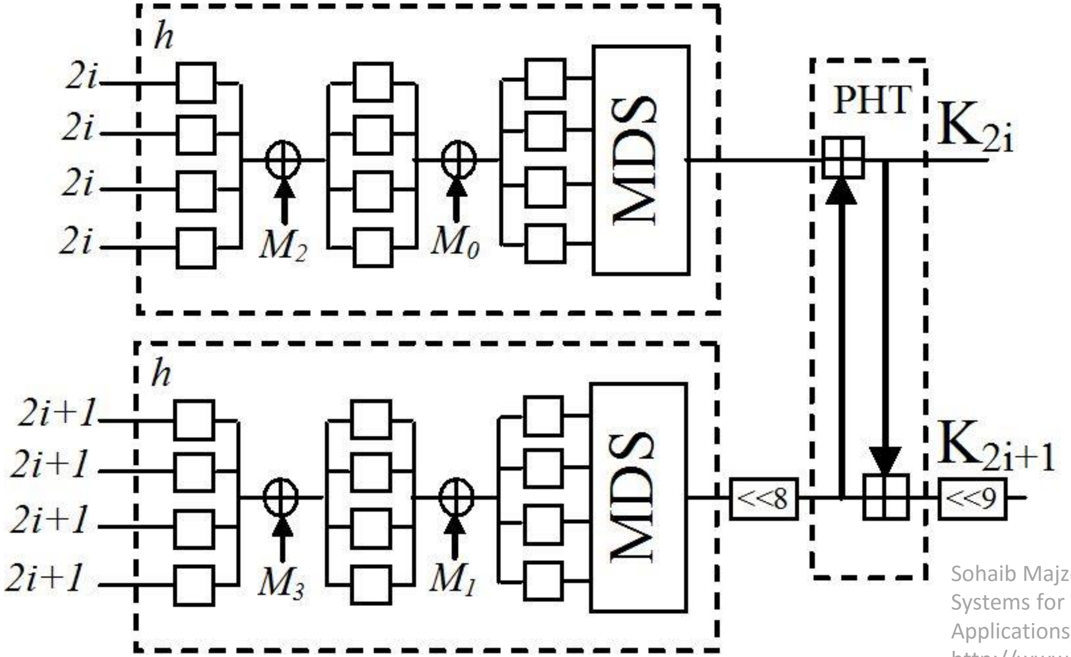
Bijective Functions

- Cryptographic block ciphers!
  - AES (Advanced Encryption Standard), Threefish, …

  - Must be bijective!
    - (Otherwise messages can’t be encrypted/decrypted)

AES-128 Algorithm

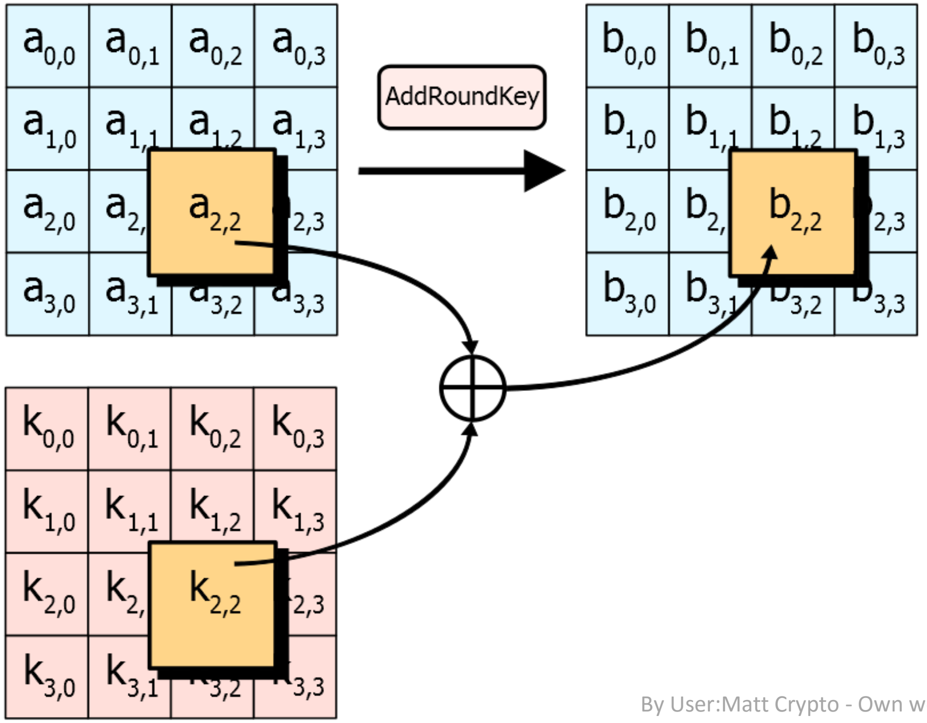
- 1) Key Expansion
  - – Determine all keys *k* from initial cipher key k_B
    - Used to strengthen weak keys



Sohaib Majzoub and Hassan Diab, Reconfigurable Systems for Cryptography and Multimedia Applications, http://www.intechopen.com/source/html/38442/media/image19_w.jpg

AES-128 Algorithm

- 2) Add round key
  - Bitwise XOR state *s* with key k_0



$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,}$	$a_{2,2}$	$_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

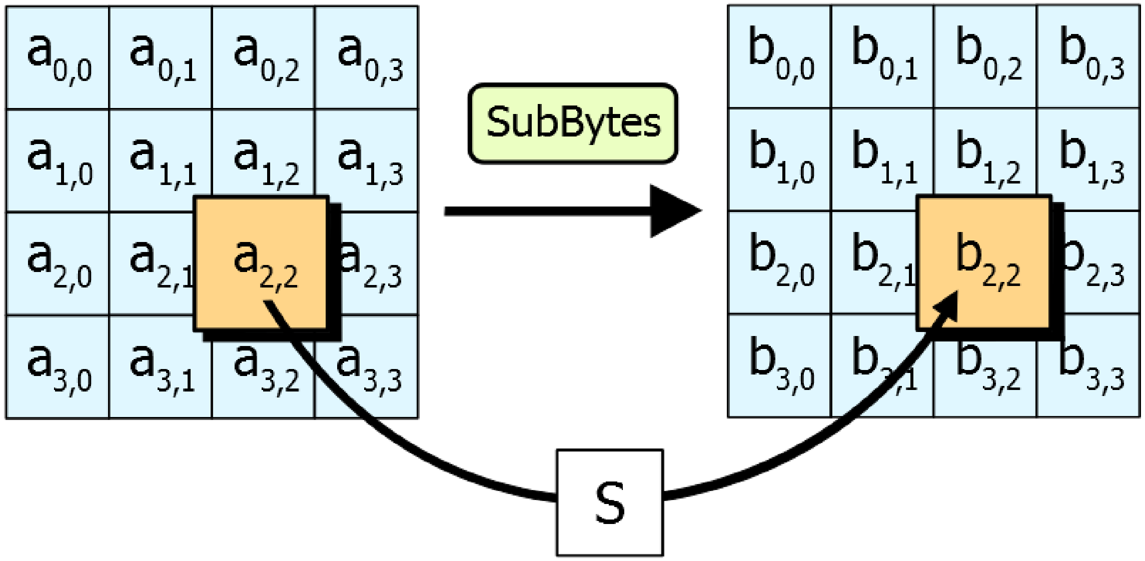
$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,}$	$b_{2,2}$	$_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,}$	$k_{2,2}$	$_{2,3}$
$k_{3,0}$	$k_{3,1}$	$_{3,2}$	$_{3,3}$

By User:Matt Crypto - Own work. Licensed under Public Domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:AES-AddRoundKey.svg#/media/File:AES-AddRoundKey.svg

AES-128 Algorithm

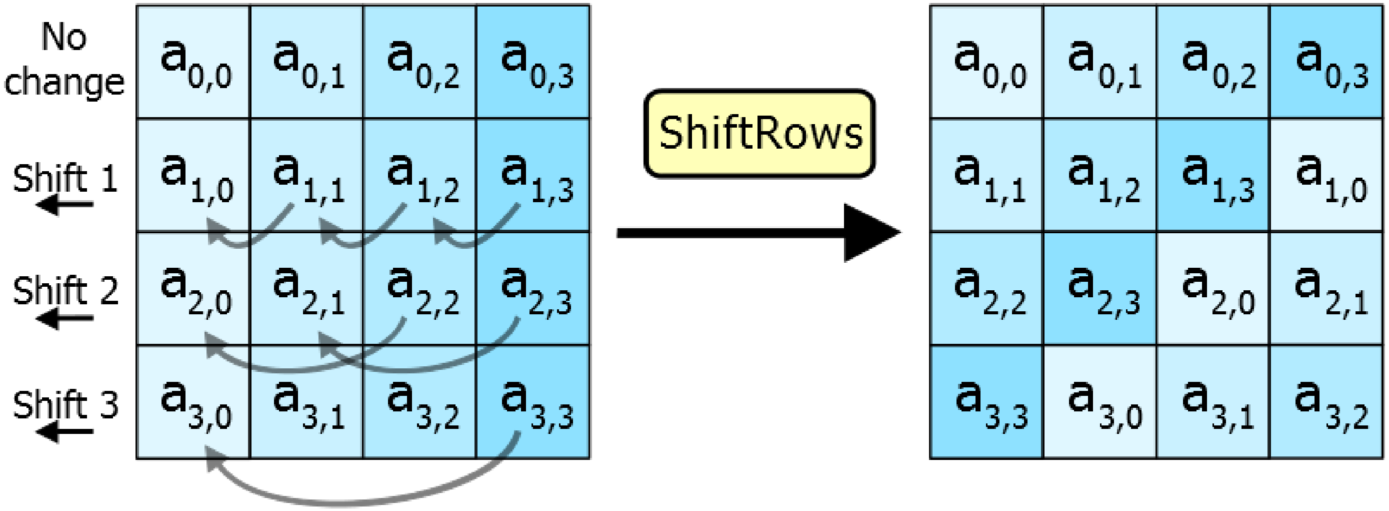
- 3) For each round… (10 rounds total)
  - – a) Substitute bytes
    - Use lookup table to switch positions


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
SubBytes
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
S

By User:Matt Crypto - Own work. Licensed under Public Domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:AES-AddRoundKey.svg#/media/File:AES-AddRoundKey.svg

AES-128 Algorithm

- 3) For each round…
  - b) Shift rows



No change	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
Shift 1 ←	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
Shift 2 ←	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
Shift 3 ←	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

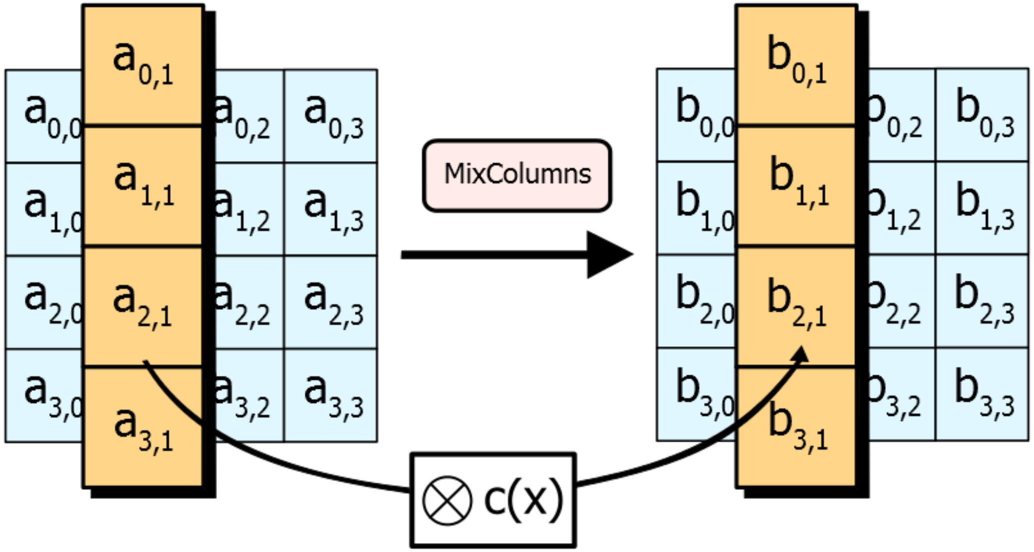
ShiftRows

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,0}$
$a_{2,2}$	$a_{2,3}$	$a_{2,0}$	$a_{2,1}$
$a_{3,3}$	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$

By User:Matt Crypto - Own work. Licensed under Public Domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:AES-AddRoundKey.svg#/media/File:AES-AddRoundKey.svg

AES-128 Algorithm

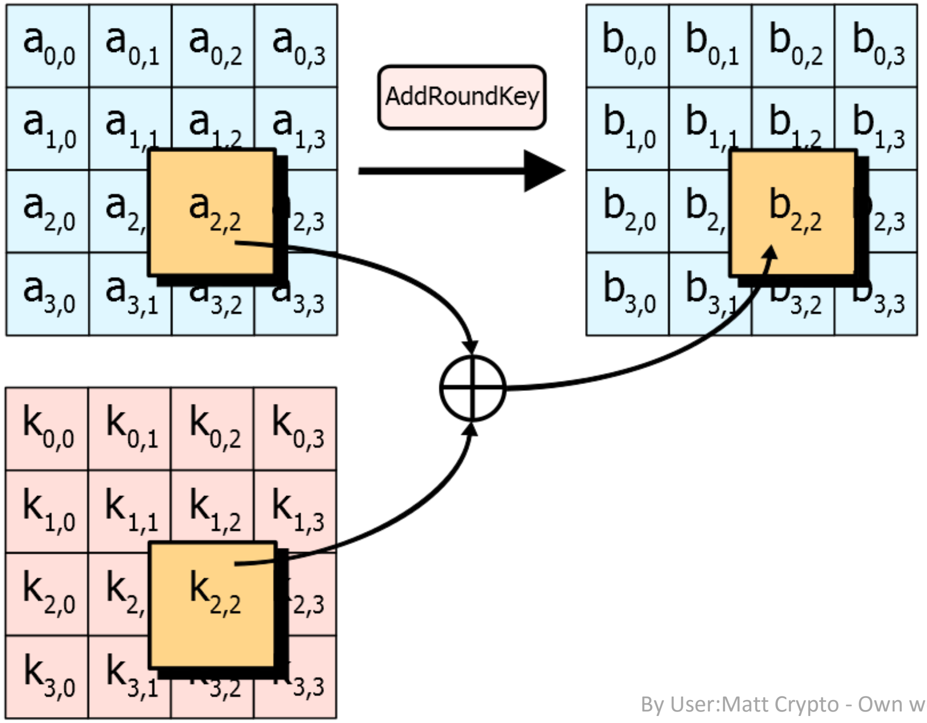
- 3) For each round…
  - c) Mix columns
    - Multiply by constant matrix $\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$



By User:Matt Crypto - Own work. Licensed under Public Domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:AES-AddRoundKey.svg#/media/File:AES-AddRoundKey.svg

AES-128 Algorithm

- 3) For each round…
  - d) Add round key (as before)



By User:Matt Crypto - Own work. Licensed under Public Domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:AES-AddRoundKey.svg#/media/File:AES-AddRoundKey.svg

AES-128 Algorithm

- 4) Final round
  - Do everything in normal round except mix columns

AES-128 Algorithm

- Summary:
  - 1) Expand keys
  - 2) Add round key
  - 3) For each round (10 rounds total)
    - Substitute bytes
    - Shift rows
    - Mix columns
    - Add round key
  - 4) Final round:
    - (do everything except mix columns)

Algorithmic Improvements

- We have a good PRNG!
  - Simple transition function *f*
    - Counter
  - Complicated output function *g*(*k*, *n*)
    - AES-128

Algorithmic Improvements

- We have a good PRNG!
  - Simple transition function *f*
    - Counter
  - Complicated output function *g*(*k*, *n*)
    - AES-128

  - High quality!
    - Passes Crush test suite (more on that later)
  - Parallelizable!
    - *f* and *g* only depend on *k*, *n* !
  - Sort of slow to compute
    - AES is sort of slow without special instructions (which GPUs don’t have)

Algorithmic Improvements

- Can we “make AES go faster”?
  - – AES is a cryptographic algorithm, but we’re using it for PRNG
  - – Can we change the algorithm for our purposes?

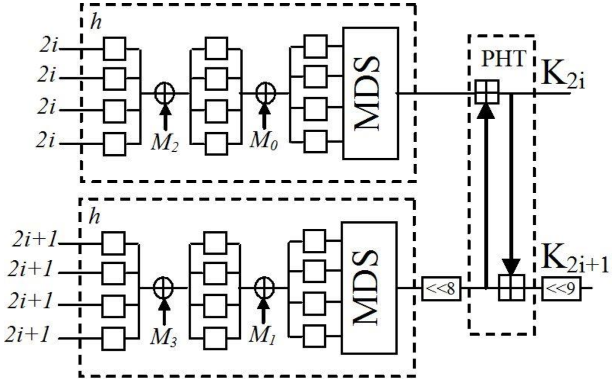
AES-128 Algorithm

- Summary:
  - 1) Expand keys
  - 2) Add round key
  - 3) For each round (10 rounds total)
    - Substitute bytes
    - Shift rows
    - Mix columns
    - Add round key
  - 4) Final round:
    - (do everything except mix columns)

AES-128 Algorithm

- Summary:
  - 1) Expand keys
  - 2) Add round key
  - 3) For each round (10 rounds total)
    - Substitute bytes
    - Shift rows
    - Mix columns
    - Add round key
  - 4) Final round:
    - (do everything except mix columns)

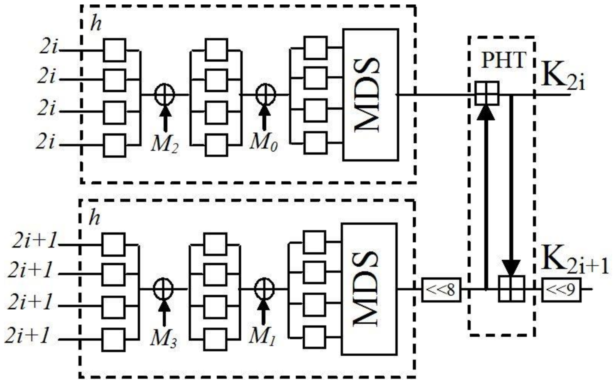
Purpose of this step is to hide key from attacker using chosen plaintext. Not relevant here.



AES-128 Algorithm

- Summary:
  - 1) Expand keys

Purpose of this step is to hide key from attacker using chosen plaintext. Not relevant here.



  - 2) Add round key
  - 3) For each round (10 rounds total)
    - Substitute bytes
    - Shift rows
    - Mix columns
    - Add round key
  - 4) Final round:
    - (do everything except mix columns)

Do we really need this many rounds?

Other changes?

Key Schedule Change

- Old key schedule:
  - The first **n** bytes of the expanded key are simply the encryption key.
  - The rcon iteration value **i** is set to 1
  - Until we have **b** bytes of expanded key, we do the following to generate **n** more bytes of expanded key:
    - We do the following to create 4 bytes of expanded key:
      - We create a 4-byte temporary variable, **t**
      - We assign the value of the previous four bytes in the expanded key to **t**
      - We perform the key schedule core (see above) on **t**, with **i** as the rcon iteration value
      - We increment **i** by 1
      - We exclusive-OR **t** with the four-byte block **n** bytes before the new expanded key. This becomes the next 4 bytes in the expanded key
    - We then do the following three times to create the next twelve bytes of expanded key:
      - We assign the value of the previous 4 bytes in the expanded key to **t**
      - We exclusive-OR **t** with the four-byte block **n** bytes before the new expanded key. This becomes the next 4 bytes in the expanded key
    - If we are processing a 256-bit key, we do the following to generate the next 4 bytes of expanded key:
      - We assign the value of the previous 4 bytes in the expanded key to **t**
      - We run each of the 4 bytes in **t** through Rijndael's S-box
      - We exclusive-OR **t** with the 4-byte block **n** bytes before the new expanded key. This becomes the next 4 bytes in the expanded key.

Copied from Wikipedia (Rijndael Key Schedule)

- New key schedule:
  - $k_0 = k_B$
  - $k_{i+1} = k_i + \text{constant}$
    - e.g. golden ratio

AES-128 Algorithm

- Summary:
  - 1) Expand keys using simplified algorithm
  - 2) Add round key
  - 3) For each round (~~10~~ 5 rounds total)
    - Substitute bytes
    - Shift rows
    - Mix columns
    - Add round key
  - 4) Final round:
    - (do everything except mix columns)

Other simplifications possible!

Algorithmic Improvements

- We have a good PRNG!
  - Simple transition function *f*
    - Counter
  - Complicated output function $g(k, n)$
    - Modified AES-128 (known as ARS-5)

  - High quality!
    - Passes Crush test suite (more on that later)
  - Parallelizable!
    - *f* and *g* only depend on *k*, *n* !
  - Moderately faster to compute

Even faster parallel PRNGs

- Use a different *g*, e.g.
  - Threefish cipher
    - Optimized for PRNG – known as “Threefry”
  - “Philox”
    - (see paper for details)
    - 202 GB/s on GTX580!
      - Fastest known PRNG in existence

General Monte Carlo method

- Pseudocode:

```
for (number of trials):
     randomly pick value from a probability distribution
     perform deterministic computation on inputs

(aggregate results)
```

What about this?

Trials are independent

Usually so
(e.g. with reduction)

- Can we parallelize this?

General Monte Carlo method

- Pseudocode:

```
for (number of trials):
    randomly pick value from a probability distribution
    perform deterministic computation on inputs

(aggregate results)
```

Yes!

Trials are independent

Usually so (e.g. with reduction)

- Can we parallelize this?
  - Yes!
  - Part of cuRAND

Summary

- Monte Carlo methods
  - Very useful in scientific simulations
  - Parallelizable because of…

- Parallelized random number generation
  - Another story of “parallel algorithm analysis”

Credits (again)

- Parallel RNG algorithm and presentation based on:
  - “Parallel Random Numbers: As Easy as 1, 2, 3”
    - (Salmon, Moraes, Dror, Shaw) at D. E. Shaw Research