Automatic Differentiation for Financial Derivatives

Antoine Savine

Introduction

- Automatic Differentiation
  - Programming technique to produce analytical sensitivities to inputs for calculation code
  - Automates the production of sensitivities
  - Achieves breathtaking speed thanks to reverse adjoint propagation (RAP)
- AD a game changer for financial derivatives
  - Risks for exotic books orders of magnitude faster
  - Risks for CVA/DVA/xVA in reasonable time
- Risks an obvious application, but with AD we can also produce:
  - Near instantaneous calibrations
  - Real-time risk for exotics
  - Combined with other techniques, ***future*** risks with Monte-Carlo
    Optimal European hedge, transaction costs, volatility bid/offers, and more
  - And more

AD and finite difference

- Finite difference
  - Bump inputs one by one and recalculate
  - Also automatic
  - Not analytical but does not matter much in practice
  - Sensitivity to n inputs costs n function evaluations
- AD
  - Calculates all sensitivities of a result in one single sweep
  - Sensitivity to n inputs is computed in constant time!

AD: benefits

- AD not entirely new in finance with pioneering work from
  - Giles & Glasserman, 2006
  - Capriotti, 2011
  - Flyger & Scavenius, 2012, *among others*
- However underused in the context of exotics and CVA with Monte-Carlo and multi-factor PDEs in most banks
- Whereas this is where AD makes the most significant difference
  - Computation is costly
  - Number of sensitivities is large
- In addition
  - AD well suited to Monte-Carlo simulations
  - AD works well with multithreading/parrallel computing
  - And is particularly well suited to development in C++

AD: limits

- AD computes sensitivities ***faster***
- But does not improve the ***quality*** of sensitivities...
  - Despite being analytical, end results typically very similar to FD with small bump
  - AD computes derivatives with constant control flow
  - Like FD, AD cannot work with discontinuous functions: digital/barrier features in Monte-Carlo
    Before application of AD/FD, function must be smoothed: Call-Spread approximation, Malliavin Calculus
- AD consumes memory
  - Consumption ~number of mathematical operations ~running time
  - On modern computers, roughly 5GB per second
  - We will review techniques to reduce consumption
- Proper implementation of AD is hard work
  - ”Automatic” differentation means final code is hassle and maintenance free...
  - ...But its efficient production takes skill and effort

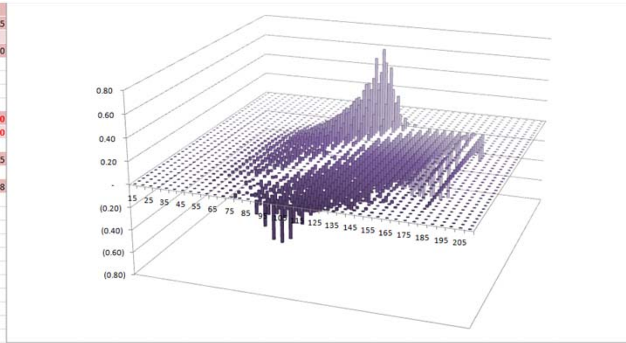
Example: Monte-Carlo Barrier Microbuckets

- Knock-out call, Monte-Carlo, local volatility model
  - Volatility function of spot and time
  - Bilinearly interpolated from local volatility matrix
  - We compute ”microbuckets” = sensitivities to all local volatilities in the matrix
- MacBook Pro 2013, quad 2.60Ghz

Microbucket computation times

Simuls	Steps	Sens	Pricing/ST	FDM	AD	FDM-MT	AD-MT
20,000	50	100	0.10sec	10sec	0.50sec	2.50sec	0.20sec
20,000	50	400	0.10sec	40sec	0.50sec	10sec	0.20sec
100,000	200	1,600	2sec	~1hour	10sec	15min	2.5sec

MT	always
Batch	25
Pin	TRUE
Threads	0
Paths	100000
Steps	200
Smooth	5
MC Pric	3.08



AD risk

Price	3.08
Delta	0.03

	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
0.25	-	-	-	-	-	-	-	-	0.00	(0.00)	0.00	(0.00)	0.00	(0.00)	0.00	(0.12)	(0.21)	(0.31)
0.50	-	-	-	-	-	-	0.00	0.00	0.00	(0.00)	(0.00)	(0.00)	(0.03)	0.00	0.02	(0.06)	(0.14)	(0.12)
0.75	-	-	-	-	-	0.00	0.00	0.00	0.00	(0.00)	0.00	(0.01)	(0.02)	(0.02)	(0.03)	(0.04)	(0.07)	(0.11)
1.00	-	-	-	-	-	(0.00)	0.00	0.00	0.00	0.00	(0.00)	0.01	0.00	0.02	(0.06)	(0.09)	(0.11)	(0.09)
1.25	-	-	-	0.00	(0.00)	(0.00)	(0.00)	0.00	0.01	0.00	0.01	0.00	0.03	0.01	(0.06)	(0.03)	(0.03)	(0.17)
1.50	-	-	-	0.00	0.00	0.00	0.00	0.00	0.01	(0.00)	0.01	0.00	0.01	(0.01)	(0.04)	(0.07)	(0.06)	(0.07)
1.75	-	-	-	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.01	0.01	(0.01)	(0.05)	(0.04)	(0.05)	(0.06)	(0.05)

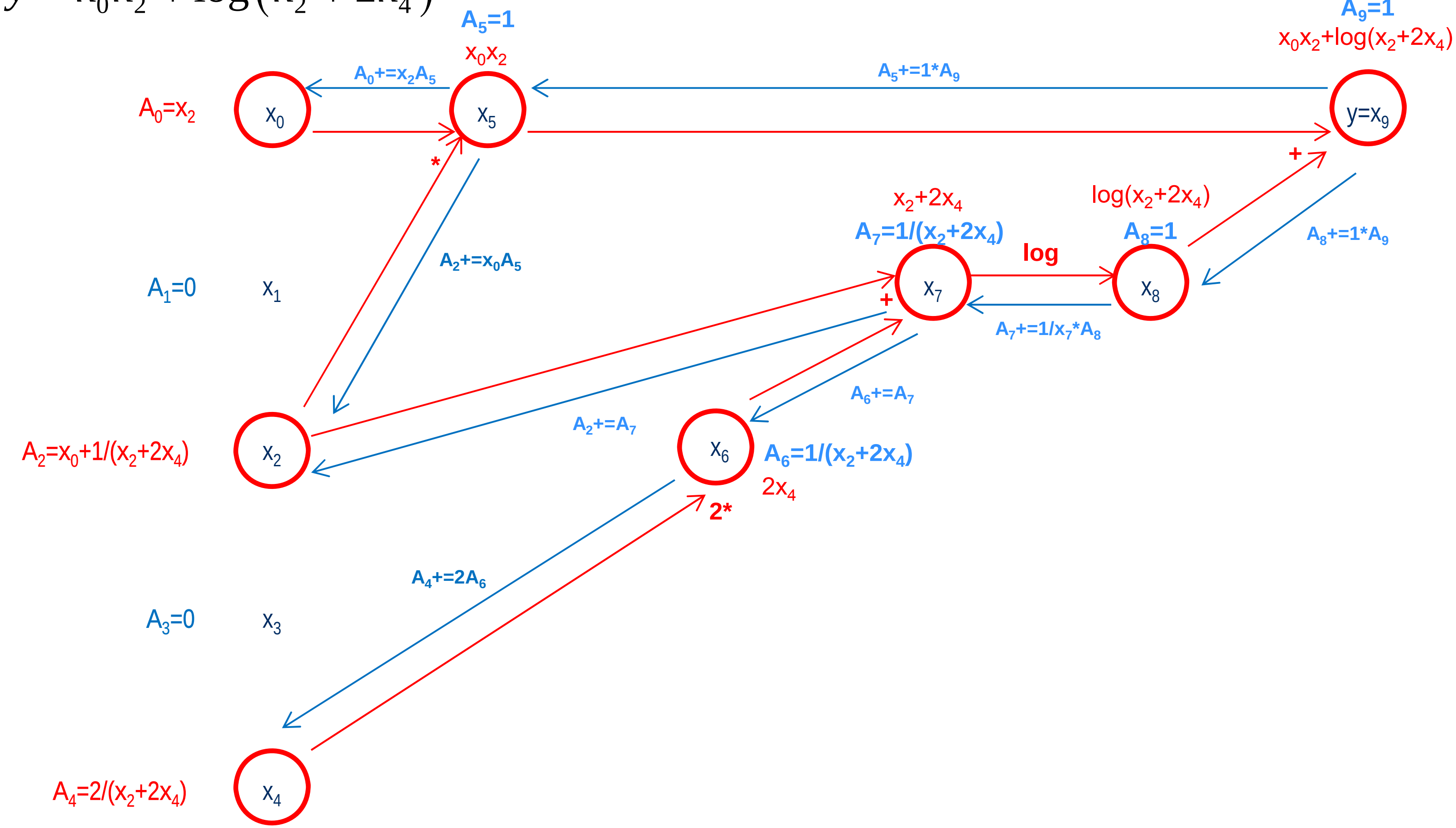
- Quick demo

Reverse Adjoint Propagation

- Any calculation program (for a given control flow) decomposes into
  - Inputs $x_0, \ldots, x_{n-1}$
  - Elementary operations: +,-,*,/,pow,log,exp,sin,etc. $i \geq n, x_i = f_i\left(x_j\right), j < i$ or $x_i = f_i\left(x_j, x_k\right), j,k < i$
  - Eventually a result $y = x_{N-1}$
  - Note: this is a decomposition per elementary operation, not per variable
- We call adjoints $A_i = \dfrac{\delta y}{\delta x_i}$
- We have
  - From the chain rule $A_i = \sum_{j \in E_i} \dfrac{\delta x_j}{\delta x_i} A_j, E_i = \left\{ j > i, \dfrac{dx_j}{dx_i} \neq 0 \right\}$
  - And obviously $A_{N-1} = 1$
- This is evaluated in reverse order from A_{N-1} to A_0
  - $\delta x_j / \delta x_i$ are known analytically, note they depend on the $\left(x_i\right)$s
  - In one single sweep where adjoints are deduced from one another

Reverse Adjoint Propagation: example

$$y = x_0 x_2 + \log\left(x_2 + 2x_4\right)$$



Reverse Adjoint Propagation: methodology

- First we performed the usual forward calculation
  keeping track of all operations
  partial derivatives depend on arguments
  ➔ *also keep track of all intermediate results*
- Then we propagated adjoints backwards through the operation chain
  All adjoints were calculated from one another in a single sweep
- Complexity
  - Forward calculation: 1x
  - Backward propagation: 1x++
  - Storage, traversal
  - Total: 4x-8x, constant in number of sensitivities

AD with operator overloading

- Use a custom type to represent numbers in place of native types (double)

```
Class adDataType{
…
};
```

- Overload all mathematical operators +,-,*,/ and functions log, exp, sqrt, etc. to:
  - Perform the calculation as for native types
  - **Record the operation and store its result**
- So that we can later run RAP throughout the operation chain

- We call ”tape” the structure in memory where operations are recorded
- And ”tape entry” each record in the tape

AD with operator overloading (2)

- In practice we also need
  - Comparison operators
  - Constructor from native types

- Template calculation code so that it can be run with the custom number type

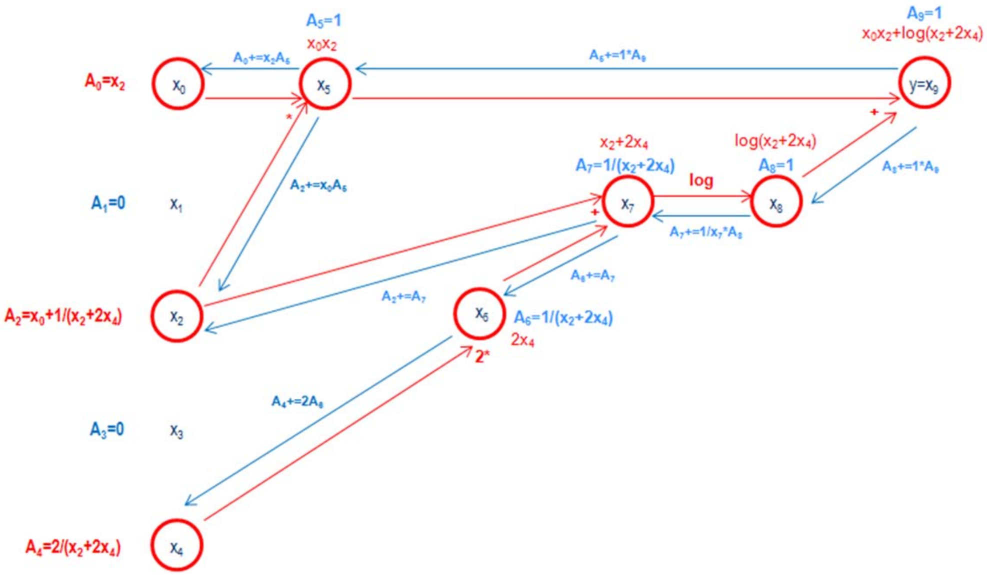
```
double calcFunc( const double x[]) {
double temp;
...
}
```

```
template <class T>
T calcFunc( const double T[]) {
T temp;
...
}
```

Danske Markets

AD programming guidelines

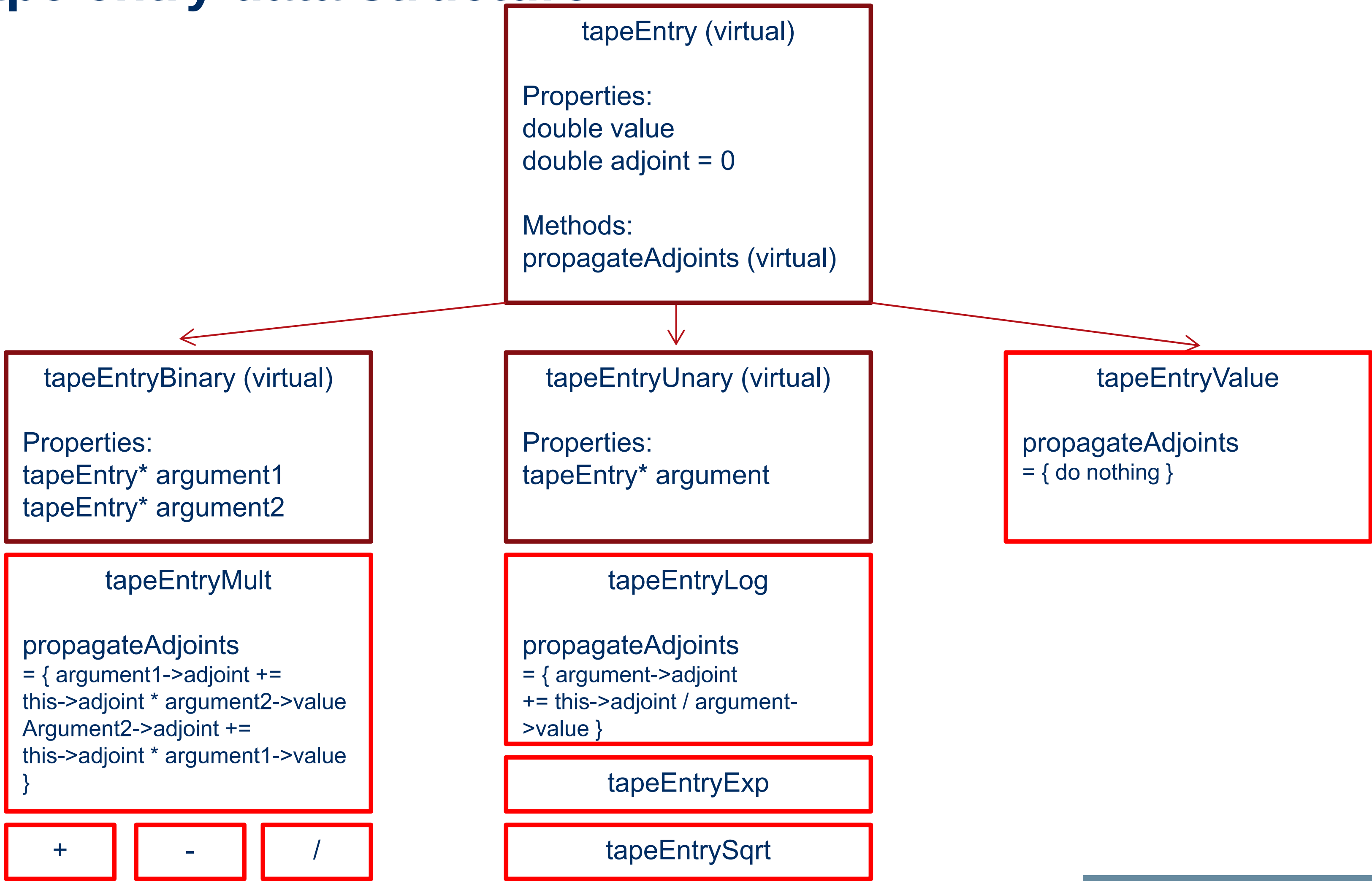
- Many possible choices, we present one
- Custom number type stores a reference to the corresponding tape entry
- Tape
  - Is global/static so as to be accessible by calculation code
  - Is allocated by blocks to avoid costly allocation for each operation
- Each tape entry stores:
  - The operation type
  - The intermediate result
  - The adjoint
  - Pointers on tape entries of the arguments
  - Everything to do with the tape entry
    must be ***overoptimized***
    because we have one for every operation



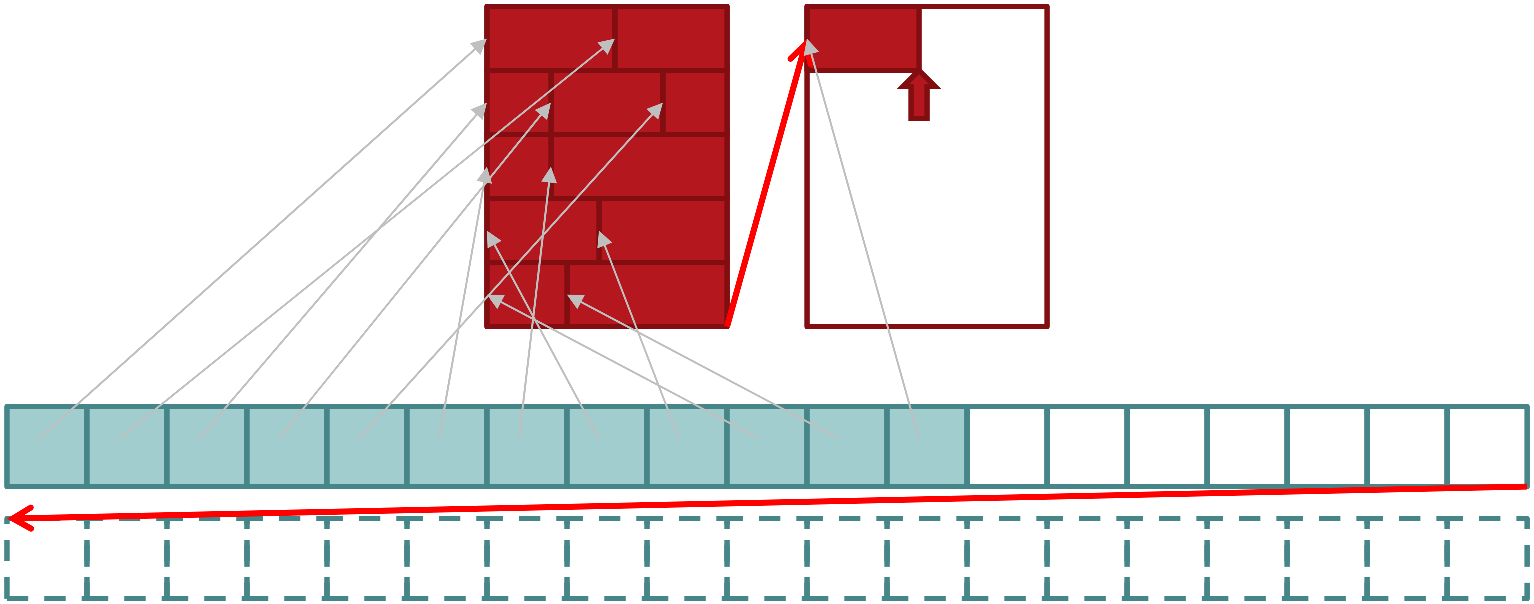
AD programming guidelines (2)

- Templated calculation code is called with the custom data type
  - Starting with an empty tape
  - Computation is performed forward
  - Operations and results are all recorded on the tape
- After calculation, we use the recorded tape
  - Adjoints are propagated backward through the tape, from last entry to first
  - Derivatives are picked as the adjoints in the relevant entries
  - Tape is wiped

Tape entry data structure



Tape data structure



- Pre-allocated blocks of memory where entries are stored
- List of references to where individual entries are stored
- Pointer to first available storage slot

Custom number data structure

- Holds no data, only reference to corresponding tape entry
- Operators/functions overloaded to store new entry in the tape

```
friend adDataType operator+( const adDataType& lhs, const adDataType& rhs)
{ return adDataType( *new ( tape()) adTapeEntryAdd( lhs.myEntry, rhs.myEntry)); }
```

return into variable holding result

construction out of the new tape entry

placement new

new tape entry

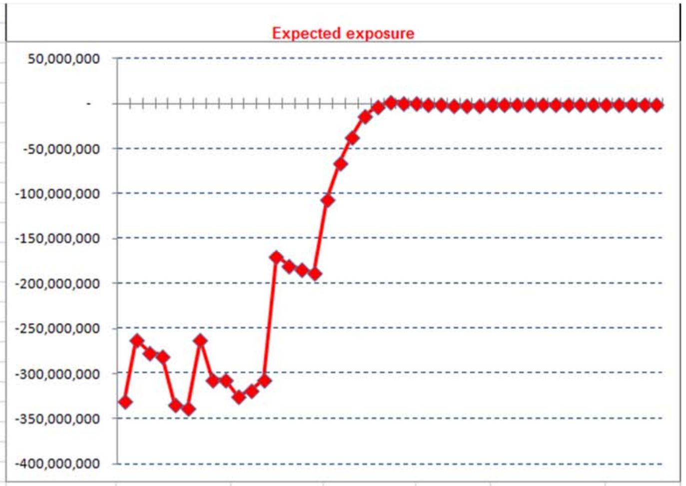
tape entry will calculate the result (sum) and store it

AD production demo

Anchor	28-Jan-14	*Pre-simulations (Longstaff-Schwartz)*	*512*
Model	BEAST	*Simulations*	*4,096*
Instrument	READCVA		
Use MT	TRUE	*Exposure quarterly over 2014, then semi-annual*	
Time	*00:00:19.24*		

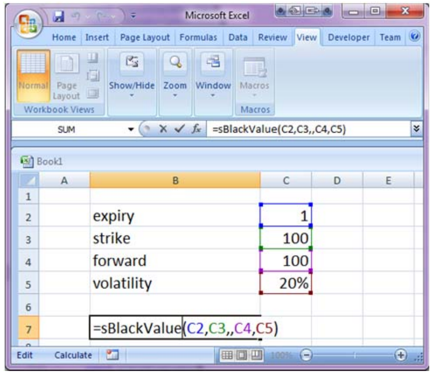
Results

variable names	values	value errors	scenario name	bump	type
CVA_NTL	11.72%	0.05%			
CVA_VAR	5,152,750	126,671			
DVA_NTL	6.10%	0.02%			
DVA_VAR	- 12,215,077	67,608			
FND_NTL	6.41%	0.02%			
FND_VAR	- 12,436,753	68,463			
ID22140859C	2,261	4,126			
ID22140862C	- 4,273,618	8,370			
ID24908088C	- 74,179	12,712			
ID24908117C	- 2,618,265	15,480			
ID25366121C	- 46,440	3,289			
ID25366155C	- 892,373	4,728			
ID25673417C	-	-			
ID26288974C	- 481,384	12,592			
ID26288980C	- 1,097,466	12,987			



Memory

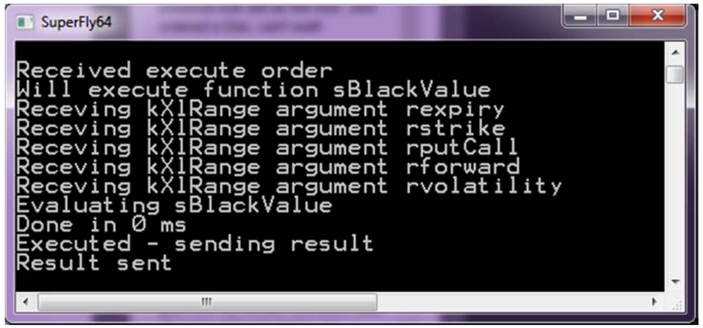
- Buy more memory!
- Note for xll users
  - 32bit Excel limits xll code memory to ~1GB
  - Get around by wrapping calculation code in quant servers, not xlls
  - Many additonal benefits to this architecture



calculation request

function arguments

result



32bit xll

64bit quant server

Performance

- Speed: expect 4x-8x the cost of calculation
  - 10 sensitivities: ~FD
  - 100 sens: up to 20x speedup
  - 1,000 sens: up to 200x speedup
- Memory: expect 5GB per second per core
  - Some algorithm (say multi-factor PDE) taking 10sec on one core
  - Will consume 50GB of memory with AD
  - With multi-threading over 10 cores, calc. time may be reduced to up to 1sec
  - But AD still consumes 50GB (5GB on each core)
  - And if calculation takes 30sec to start with, AD will consume ~150GB of memory!
- **Plus the real memory constraint is the size of the cpu cache**
  - Propagation through large tapes produces constant copying in and out of the cache
  - Substantial performance drag

Checkpointing

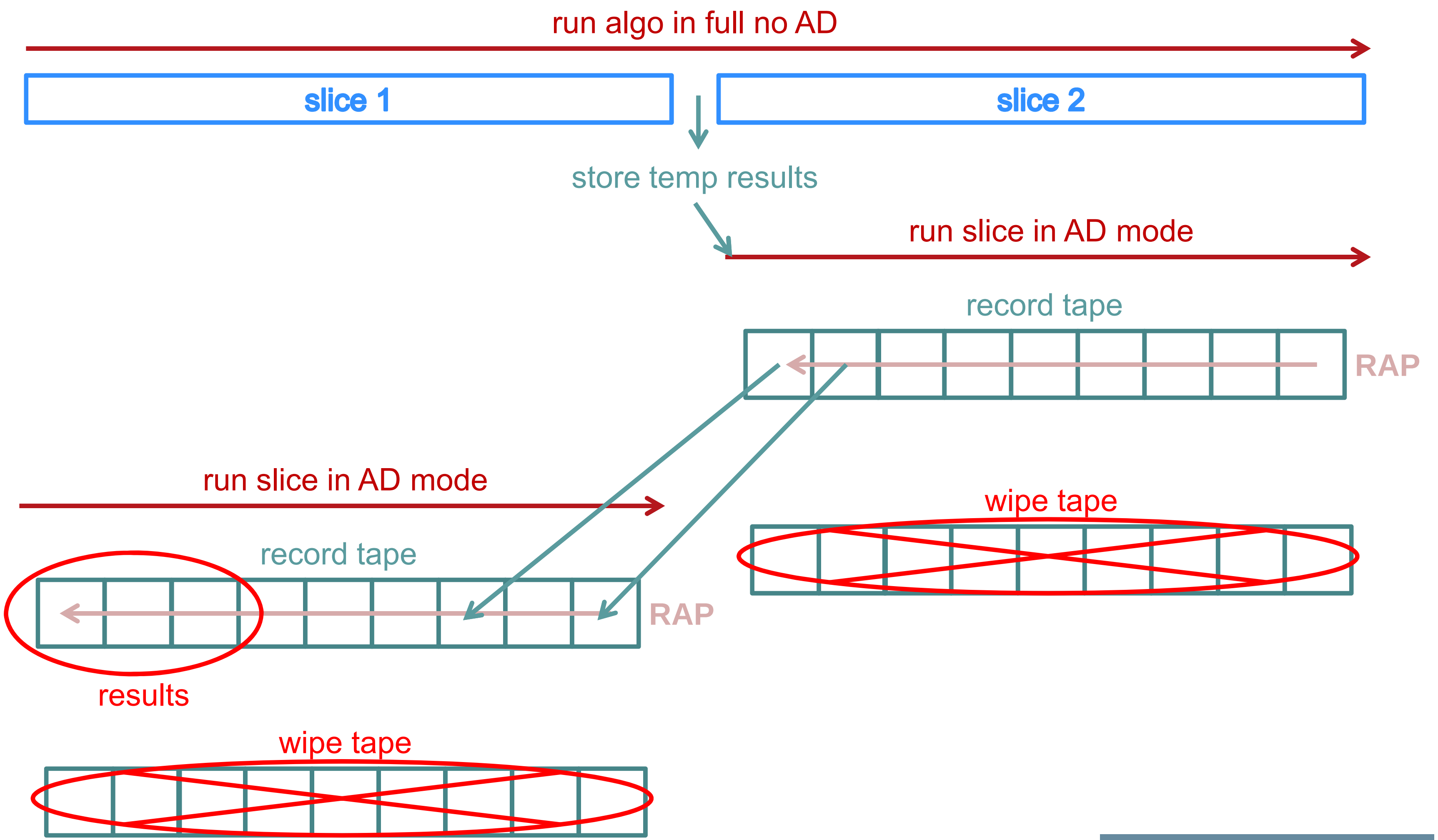
- Cache optimization
  - Work with many small tapes
  - Avoid long tapes
  - Easier said than done?
- Monte-Carlo
  - Calculate derivatives pathwise, then average
  - Tape records only one path, typically (much) less than 1/100s or 50MB per core
- What about PDE?

Checkpointing (2)

- General technique called ”checkpointing”
  - Split algorithm in slices
  - Use a tape for each slice
  - Wipe the tape between slices
  - Aggregate derivatives across slices

- Cut and paste tape entries
  - General idea: RAP works with any boundary condition, does not have to be 1 for result, 0 elsewhere
  - Hence, we ”paste” adjoints resulting from RAP on slice n as boundary conditions for slice n-1
  - A general form of checkpointing that works with PDE and a lot more

Checkpointing (3)



AD for multithreaded algorithms

- We need each thread to have its own tape
- Access to tape must be global/static so that every overloaded operation may use it
- Solution (in C++): make it *thread_local*
  - thread_local variables work like global/static variables for all intents and purposes
  - However each thread has its own copy
  - Exactly fits our needs
  - Support for thread local variables in C++11 standard (Visual Studio 12+), Boost, Windows API, ...
- Hence making AD work with MT algos is as easy as
  - Allocate a tape for each thread
  - Mark the tape accessor as thread local

Only instrument active code

- Instrument code = use custom number type
  - Operations are recorded
  - Incorporated in sensitivities
- Not instrumented code = use native number type
  - Operations are ***not*** recorded
  - Ignored for sensitivities
- Opimizations
  - Do not instrument ***inactive*** variables: variables that do not depend on inputs

    Example: uniform and gaussian random number generation for Monte-Carlo
  - Do not instrument ***active*** variables which effect on sensitivities is best ignored

    Example: size of PDE grid, depends on volatility, however d(result) / d(size) is numerical noise

    Example 2: LSM, see next

Why LSM needs no instrumentation

- The PV of a set of cash-flows early exerciseable at some date T_i

$$f_i = E\left[PV_i 1_{\{PV_i > 0\}} \right], PV_i = E_{T_i}\left[\sum_{j \geq i} DF\left(T_i, T_j\right) CF_j \right]$$

- Through LSM $f_i = E\left[PV_i 1_{\{proxy(PV_i) > 0\}} \right]$

- Hence $\dfrac{\partial f_i}{\partial a} = E\left[\dfrac{\partial PV_i}{\partial a} 1_{\{proxy(PV_i) > 0\}} \right] + E\left[PV_i \dfrac{\partial 1_{\{proxy(PV_i) > 0\}}}{\partial a} \right]$

pathwise diff with frozen exercise boundary

=0 if proxy is good

- If proxy is bad, rhs is not 0 but it is numerical noise we want to ignore
- Hence LSM is ***inactive*** and does not require instrumentation

Optimization with expression templates

- AD still in active development with very recent imporvements
- 2014 Paper from Robin J. Hogan, University of Reading
  - ”Fast reverse-mode automatic differentiation using expression templates in C++”
  - Store tape entries per expression, not per operation
  - For instance $y = x_0 x_2 + \log\left(x_2 + 2x_4\right)$ uses 1 tape entry
  - Tape entries are multinomial: n references to arguments, as opposed to max 2
  - Tape entries do not store operation type, but directly sensitivities to arguments
  - Dimension and all partial derivatives figured at compile time using *expression templates*
- The goal is to reduce the number of tape entries for a given code
  - Same number of calculations
  - But a lower number of tape entries
  - Hence a reduction in memory usage and time spent in tape traversal
  - Improvement of 10 to 50% depending on the ratio of expressions to operations in the code

Questions, comments, suggestions most welcome

adCentral@aSavine.com