

KLU and PSS Analysis Implementations into NGSPICE



Francesco Lannutti^{1,2}, Stefano Perticaroli^{1,2}

¹ University of Roma "Sapienza", DIET

² NGSPICE Project



SAPIENZA
UNIVERSITÀ DI ROMA

NGSPICE People

Italian Wing

Francesco Lannutti

Stefano Perticaroli

Paolo Nenzi

German Wing

Dietmar Warning

Holger Vogt

Robert Larice

Overview

A short
introduction to
SPICE

SPICE Challenges

SPICE Algorithm

KLU
integration into
NGSPICE –
New Version

KLU data structures

The Binding Table Idea

Performance Analysis and Conclusion

PSS
implementation
into NGSPICE

Transient Shooting Algorithm

Autonomous Case Implementation

Examples

Appendix:
KLU Sparse
Direct Linear
Solver

Properties of the Circuit Sparse Matrices (Davis)

How KLU works

SPICE Challenge

SPICE3 transient analysis performances

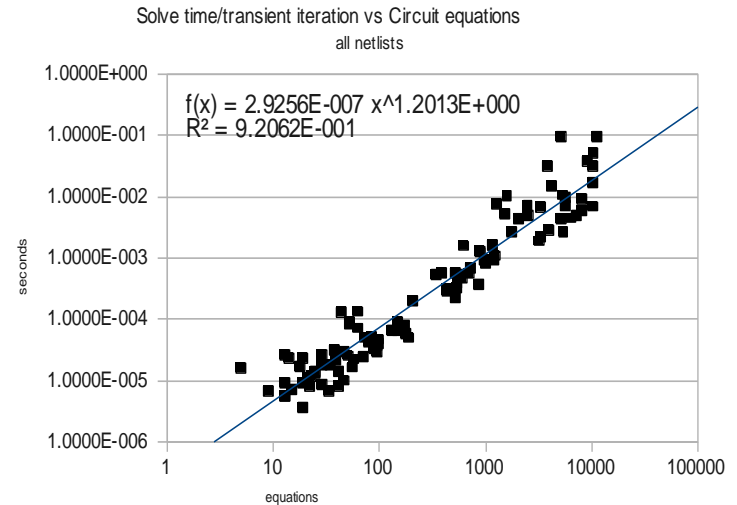
- The processing power of CPUs scales as $N^{0.96}$
- The total solve time of spice scale as $N^{1.2}$
- New technologies are described increasing models complexity
- **As technology improves, SPICE will become slower in simulating denser circuits**

Actual performances over number of circuit equations:

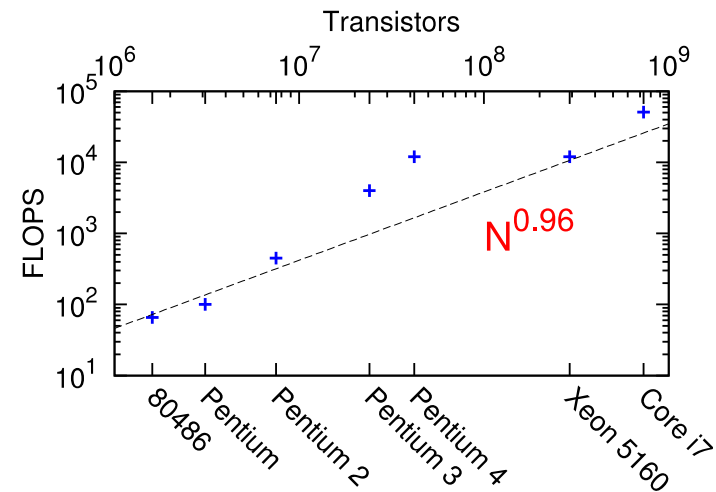
Total analysis time $8.75E-7 * N^{1.3}$

Total factor time $4.36E-7 * N^{1.4}$

Total solve time $2.93E-7 * N^{1.2}$



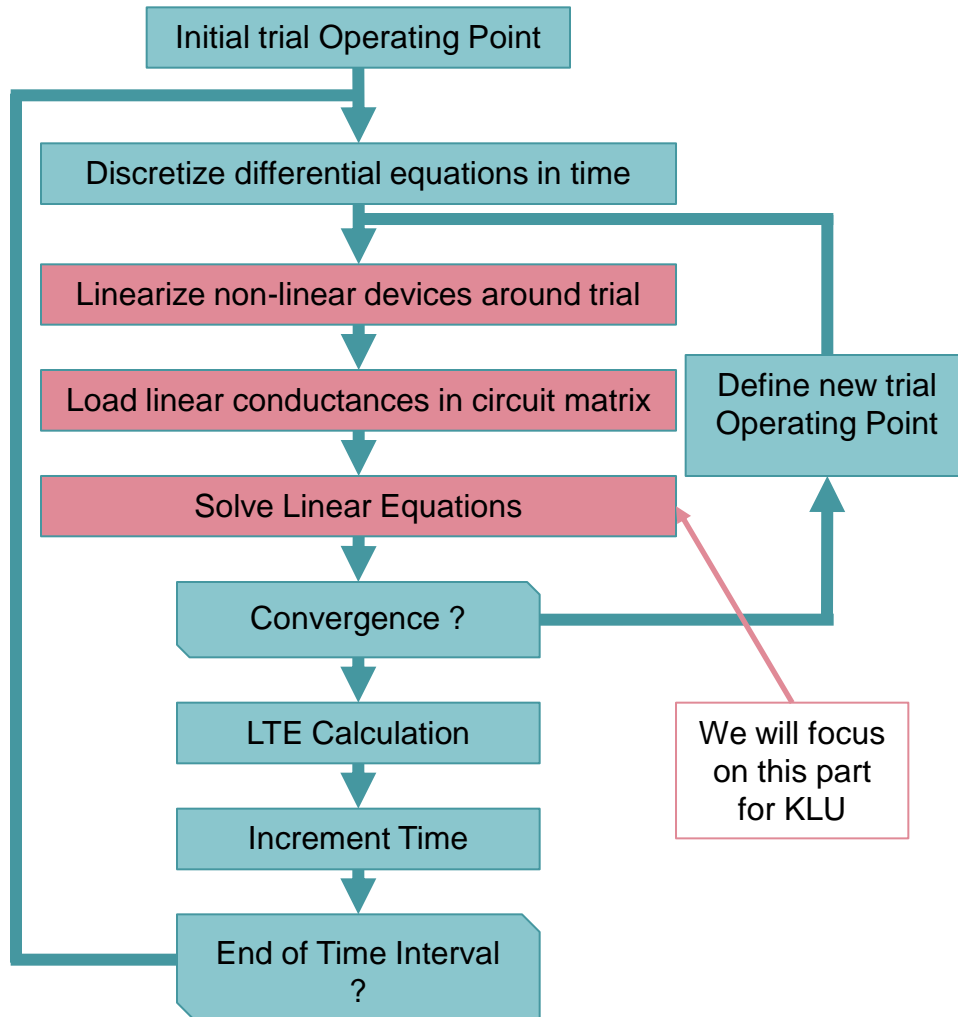
Delitala V, Bachelor's Thesis, 2009



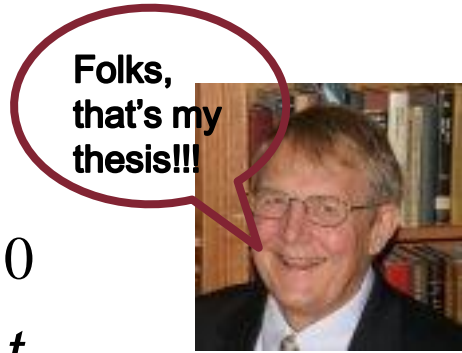
Kapre N, PhD Thesis, 2010

SPICE Algorithm

SPICE3 transient analysis algorithm



We will focus on this part for KLU



$$\mathbf{F}(\dot{\mathbf{x}}, \mathbf{x}, t) = 0$$

$$h_{n+1} = t_{n+1} - t_n$$

$$\dot{\mathbf{x}}_{n+1} = \sum_{i=0}^k \dot{\mathbf{a}}_i \mathbf{x}_{n-1} + h_{n+1} \sum_{i=-1}^k \dot{\mathbf{a}}_i \mathbf{b}_i(\mathbf{x}_{n-i}, t_{n-i})$$

$$\mathbf{x}_r^{m+1} = \mathbf{x}_r^m - a(\mathbf{x}_r^m) \frac{\mathbf{g}(\mathbf{x}_r^m)}{\mathbf{g}'(\mathbf{x}_r^m)}, 0 < a \leq 1$$

$$\mathbf{J}(\mathbf{x}^m) \mathbf{x}^{m+1} = \mathbf{J}(\mathbf{x}^m) \mathbf{x}^m - \mathbf{g}(\mathbf{x}^m)$$

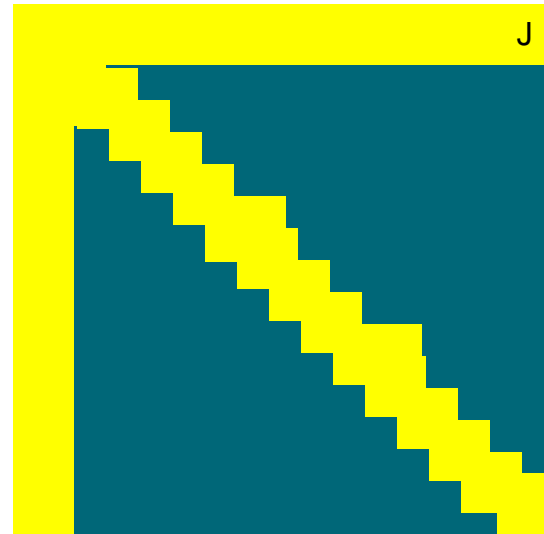
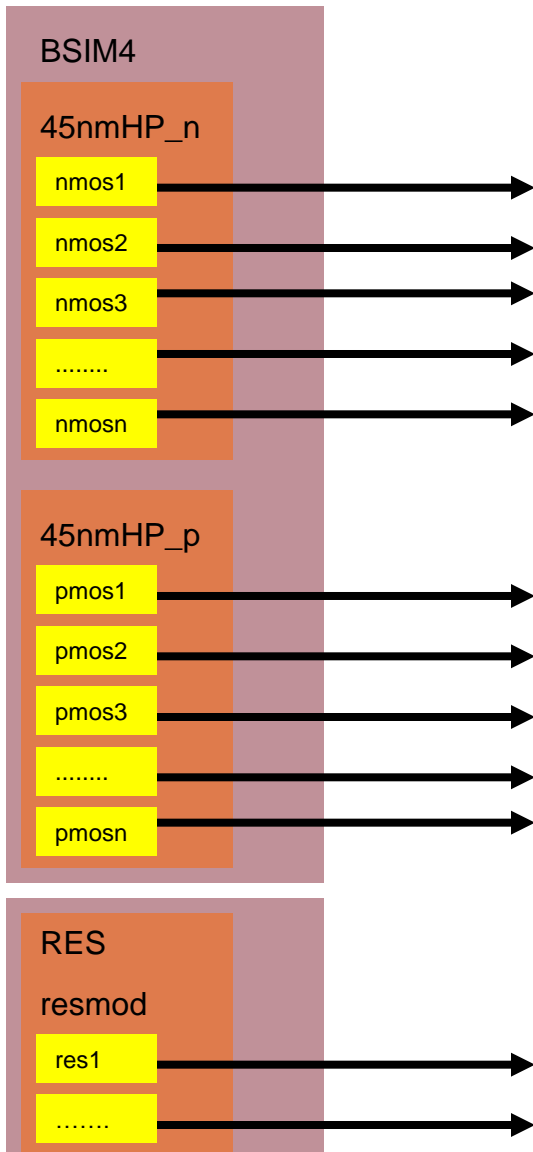
$$\mathbf{A} \mathbf{x}^{m+1} = \mathbf{b} \quad \text{where}$$

$$\mathbf{A} = \mathbf{J}(\mathbf{x}^m) + \mathbf{G}$$

$$\mathbf{b} = \mathbf{J}(\mathbf{x}^m) \mathbf{x}^m - \mathbf{g}(\mathbf{x}^m) + \mathbf{C}$$

KLU INTEGRATION INTO NGSPICE – NEW VERSION

How spice loads devices into the matrix



for each device_type (i.e. BSIM4)
for each device_model (i.e N_type)
for each instance_of_the_model
Evaluate $J(V)$ and $RHS(V)$
Load J and RHS

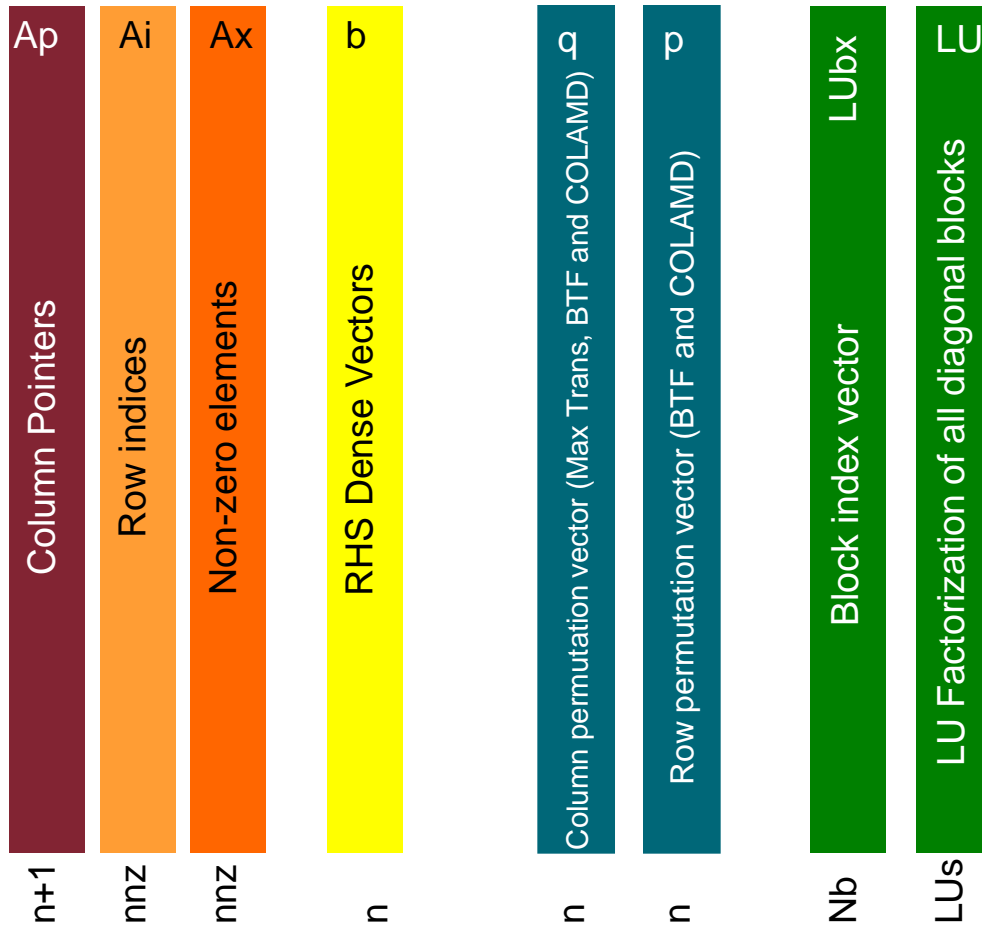


SPARSE 1.3:

- The matrix is stored as a linked list (Setup Phase)
- The matrix is built iteratively by the instances (Setup Phase)
- **Each instance knows the memory address of its elements**
- This feature gives to device code an $O(1)$ access to the matrix elements
- Matrix factorization operations alter the structure of the matrix by changing pointer relationships between elements
- ***We want to keep this, but KLU doesn't, so we need a solution!***

KLU Data Structures

KLU uses data structures based on array for both A and LU factorization



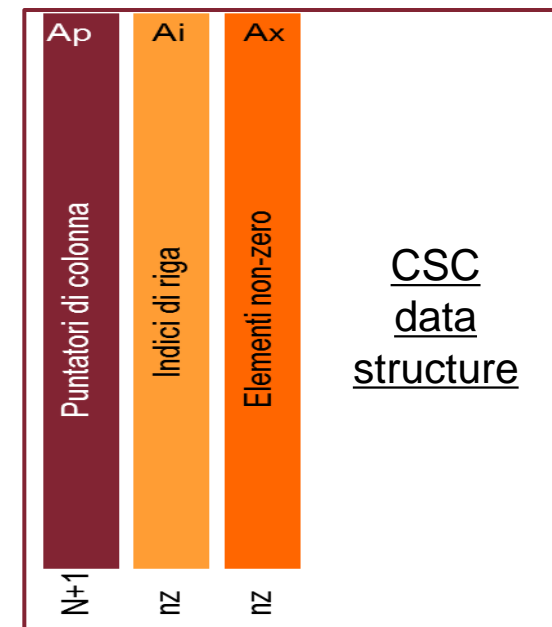
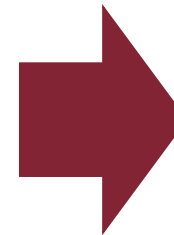
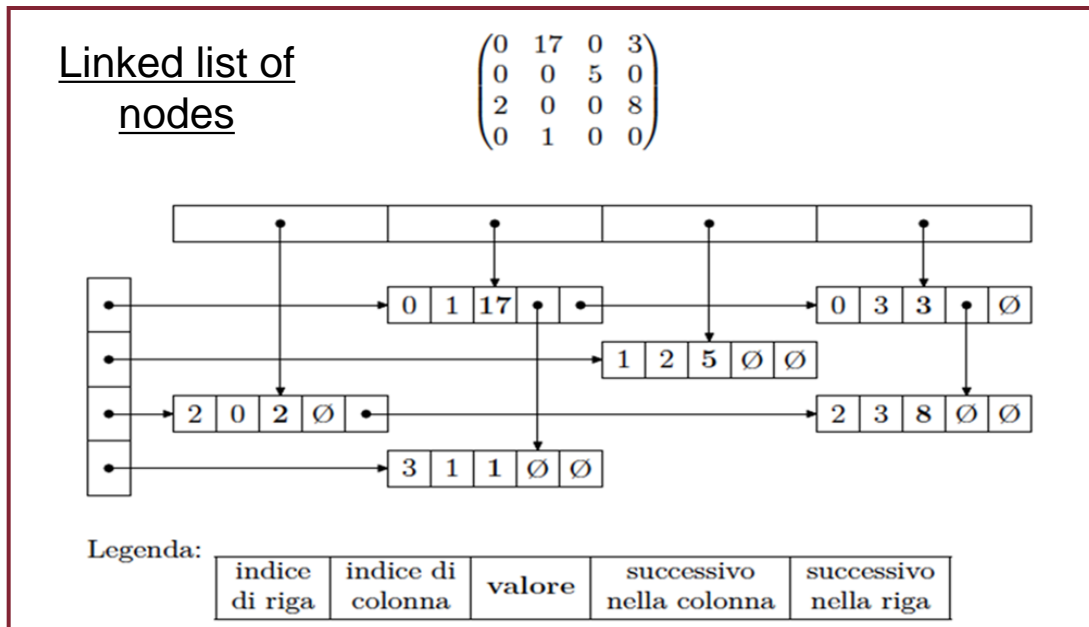
- Sparse Matrix is n-by-n
- It has «nnz» non-zero elements
- The RHS vector is dense
- The «p» and «q» vectors contain row and column permutations
- LU Factorization creates two new vectors:
 1. LUBx, that hosts the pointers to every LU structure, for every diagonal block
 2. LU, that hosts the factorization for each diagonal block

Thus KLU sacrifices space to improve time, but this is a good feature, because today space costs less than time!

KLU integration into NGSPICE

Binding solution idea to respect both SPICE and KLU requirements

- 1) Netlist Parsing
- 2) Devices Setup
- 3) Matrix Allocation as linked list of nodes – **no values written yet, only locations**
- 4) Binding Table Creation
- 5) Linked List to CSC Data Structure Conversion



Binding Table (“Associative Cache”) Example

Array Of Structure		
Sparse original address	KLU new address	KLU Complex new address
5233563	5982185	5990465
5238545	5985402	5992089
.	.	.
.	.	.
.	.	.
.	.	.
5236487	5988432	5995679

The **New Binding Table** is reordered using Quick Sort Algorithm
Every element now is searched using **Binary Search Algorithm**
instead of Linear Search Algorithm

Model Binding Example

DEVptr contains the address of an element in the matrix

```
here -> DEVptr += expression ;
```

Model code

SPARSE

```
matched = BinarySearch (DEVptr, Binding Table) ;  
here -> DEVbindStructPtr = matched ;  
here -> DEVptr = here -> DEVbindStructPtr -> CSC ;
```

DEVbindCSC routine

```
here -> DEVptr += expression ;
```

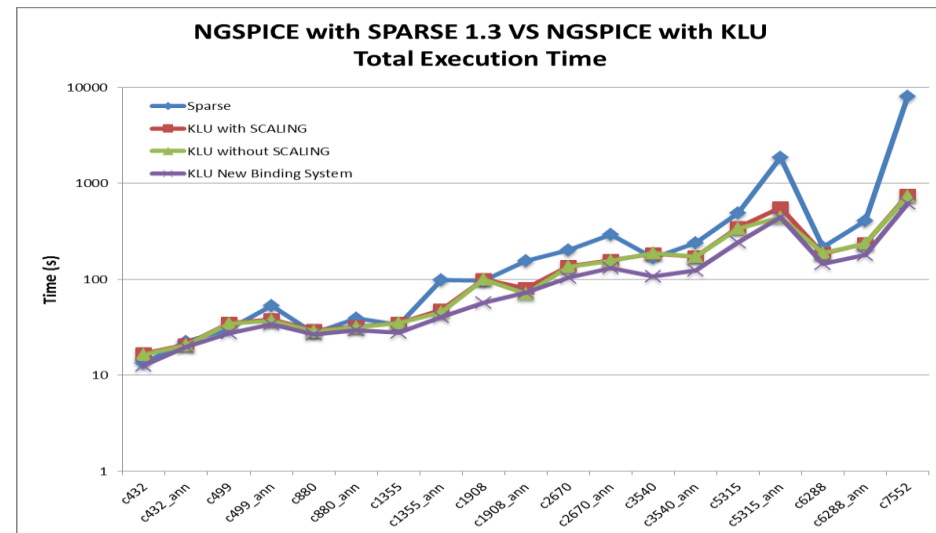
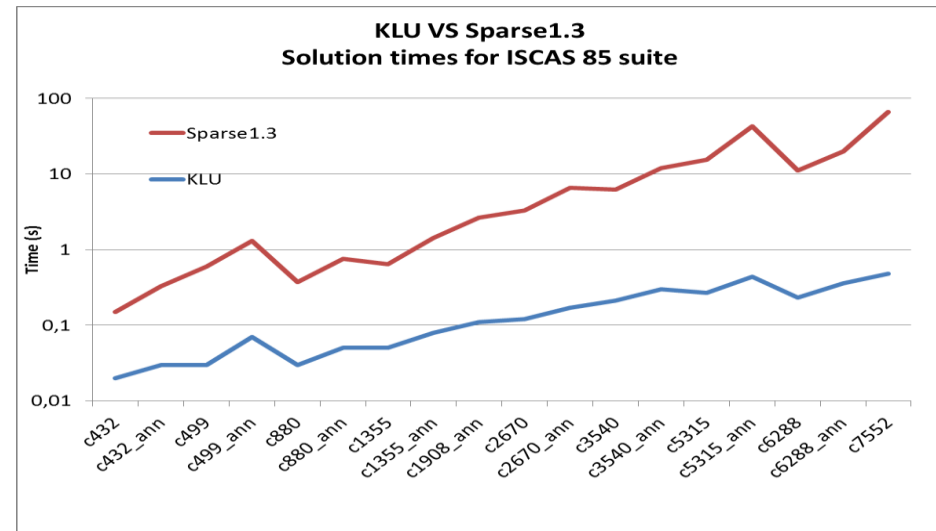
Model code

**KLU
NEW
BINDING
SYSTEM**

This New Binding System let switch between KLU (and KLU Complex) and SPARSE 1.3 representations very quickly

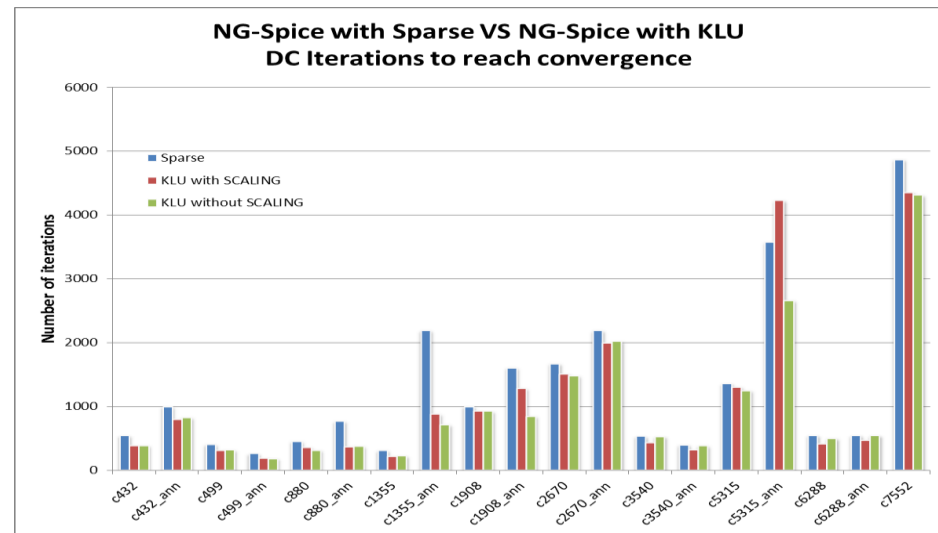
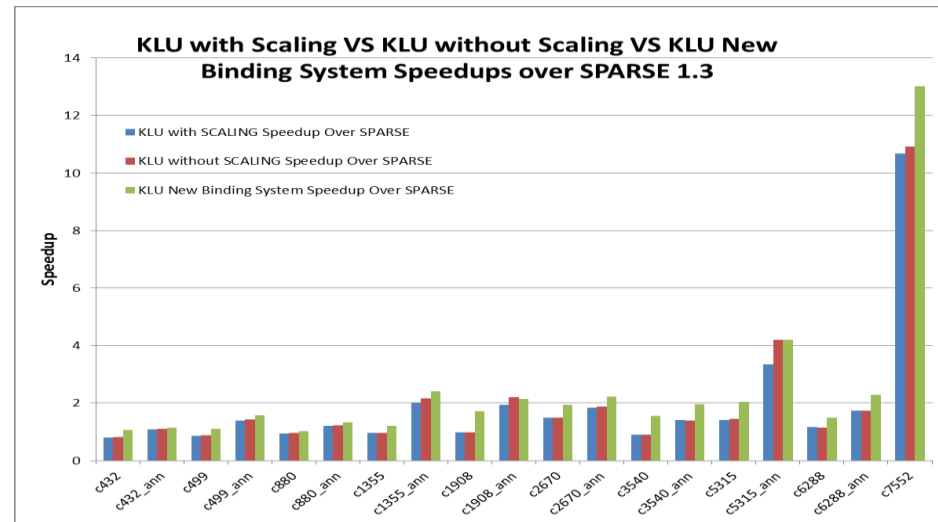
Conclusion

- KLU is faster than SPARSE 1.3 in **all matrices extracted from the ISCAS85 suite circuits**, with a speedup up to about 100x
- This very good behavior in the linear solution decreases when KLU is inserted into NGSPICE, where there are other factors needed to be accounted for timing calculation, but KLU is however faster than SPARSE 1.3
- When KLU is inserted into NGSPICE, the speedup over SPARSE 1.3 is up to about 11x. **With the New Binding System, it's up to about 13x and KLU has always faster or the same time**



Conclusion

- The difference between with scaling and without scaling is often ignorable, but sometimes scaling improves convergence, even though the total time expended is less when scaling feature is disabled. **The new version includes scaling too!**
- KLU generally improves convergence, but, even when the DC iterations count is higher than SPARSE 1.3, **NGSPICE with KLU is however faster**



Author: Stefano Perticaroli

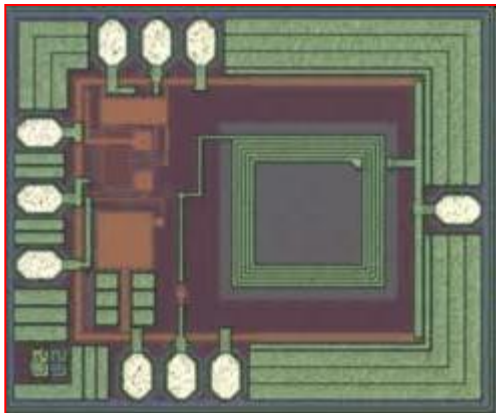
Reviewers: Francesco Lannutti and Stefano Perticaroli

PSS IMPLEMENTATION INTO NGSPICE

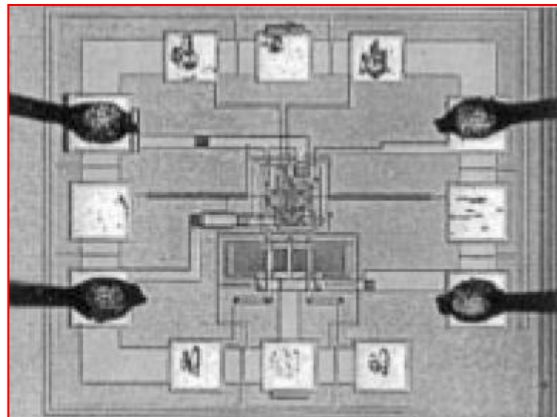
Periodic Steady State Implementation in NGSPICE

PSS is a RF dedicated analysis needed when:

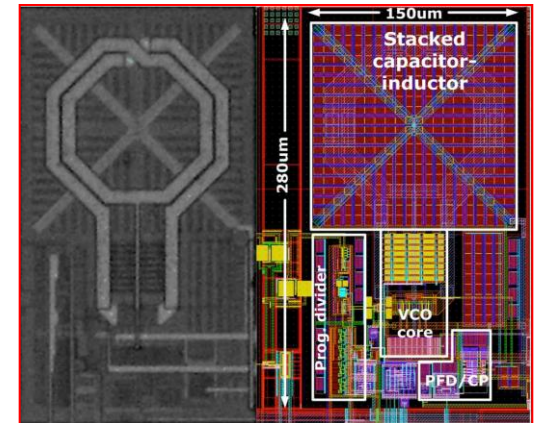
- large signal / non-linear behavior of a periodic dynamical system has to be investigated
- accurate prediction of harmonic content and harmonic related parameters are required



0.18um CMOS LNA



0.5um CMOS mixer



45nm CMOS fully integrated PLL

non autonomous circuits

autonomous circuits

PSS methods and type of circuits

PSS solution of a circuit (if exists) can be reached in general by means of:

- transient shooting (TS)
- harmonic balance (HB)
- mixed domain techniques (MDT)

preferred for

- highly non linear circuits
- large circuits

non autonomous circuits

Unknowns: nodes voltages and branches currents at PSS

Knowns: amplitudes and frequencies of driving signals and IC set

autonomous circuits

Unknowns: nodes voltages, branches currents and also working frequency at PSS

Knowns: only IC set

Transient Shooting

TS is considered in the proposed implementation

circuit system-equation in space state TD form [1]

$$f(v(t), t) = \frac{d}{dt}q(v(t)) + i(v(t)) + u(t) = 0$$

$u(t)$ input sources
 $v(t)$ nodes voltages
 $i(v(t))$ resistive currents
 $q(v(t))$ nodes charges or fluxes
Everyone is $\in \mathbf{R}^N$

PSS solution is reached when: $v(t + T) - v(t) = 0, \quad \forall t \in \mathbf{R}$

Period T is an unknown in autonomous circuits -> **no closed-form solution exists**

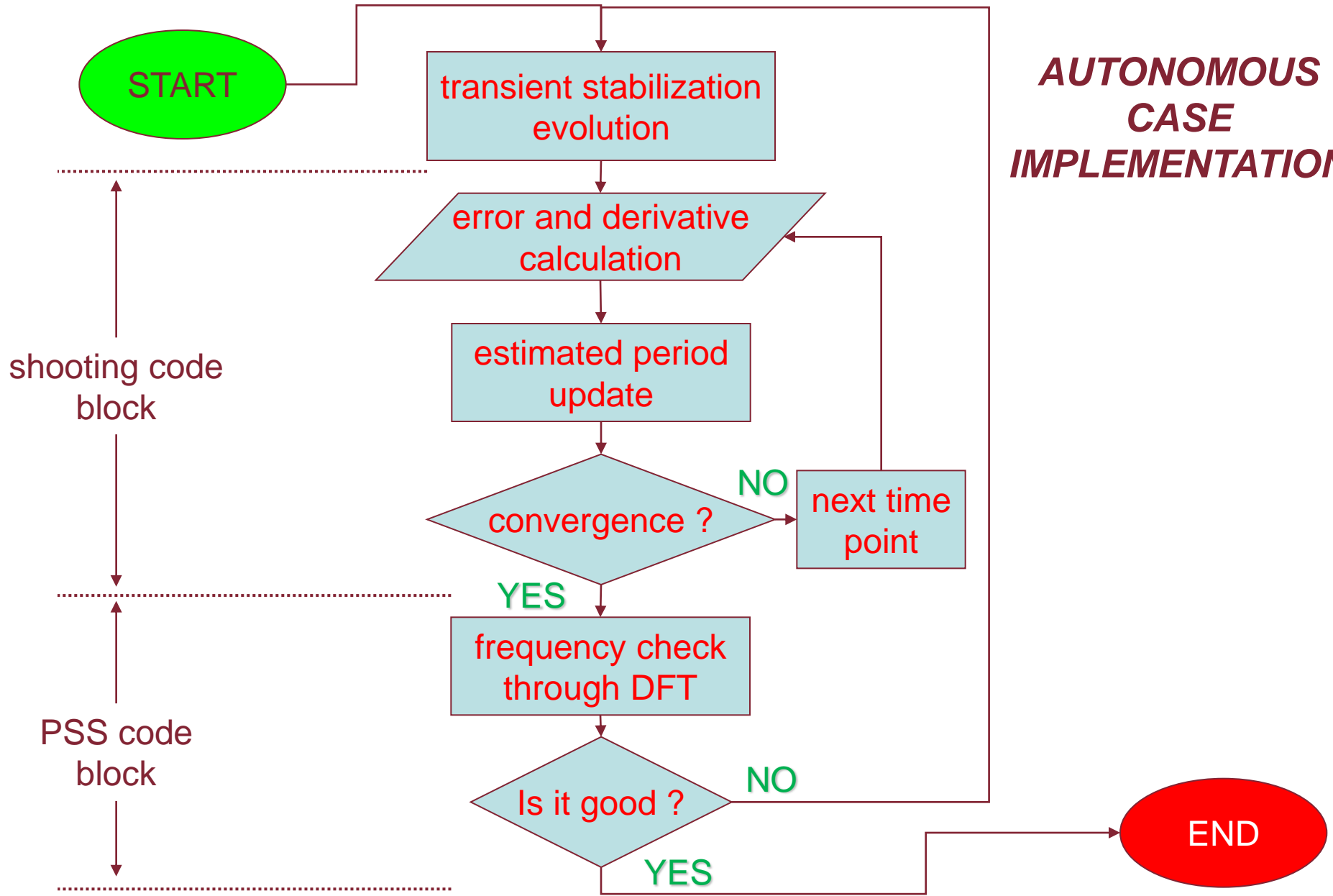
A **residual error** ε must be accepted in any numerical formulation:

$$v(\tilde{t} + \tilde{T}) - v(\tilde{t}) = \varepsilon, \quad \forall t \in \mathbf{R}$$

tilde expresses quantized quantities

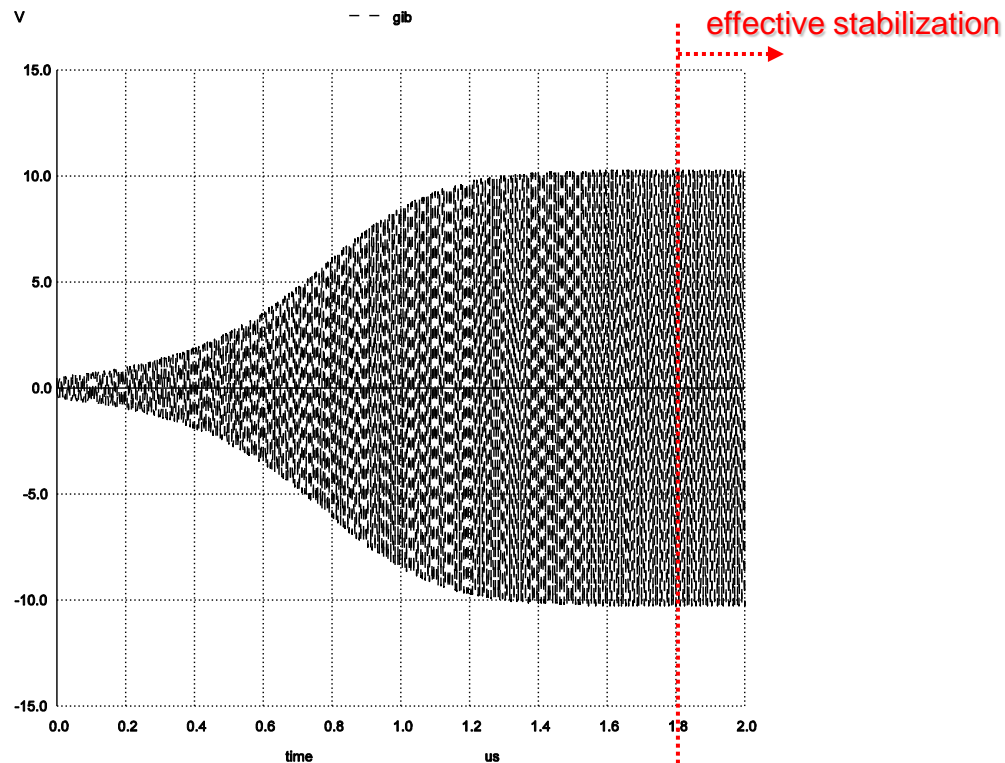
The problem becomes finding an iterative algorithm capable to decrease residual error under an acceptable tolerance. Many algorithms have already been proposed to address this problem in time domain

AUTONOMOUS CASE IMPLEMENTATION



PSS Algorithm – Stabilization Time

During the Stabilization Time we wait until the circuit becomes stable after some time, specified by the **tstab** parameter



Voltage Transient during the Stabilization Time of a VdP oscillator

PSS Algorithm – Out of Stabilization Time

Exiting from stabilization, RHS [k] is taken as reference pseudo-state vector (PSV)

MNA formulation used in NGSPICE – \tilde{t} is the transient step index

$$\begin{bmatrix} G[\tilde{t}] & B \\ C & D \end{bmatrix} RHS[\tilde{t} + 1] = RHS[\tilde{t}]$$

(N+M)x(N+M) matrix where N is the number of voltage nodes and M is the number of current branches

(N+M)x1 solutions vector

(N+M)x1 known vector

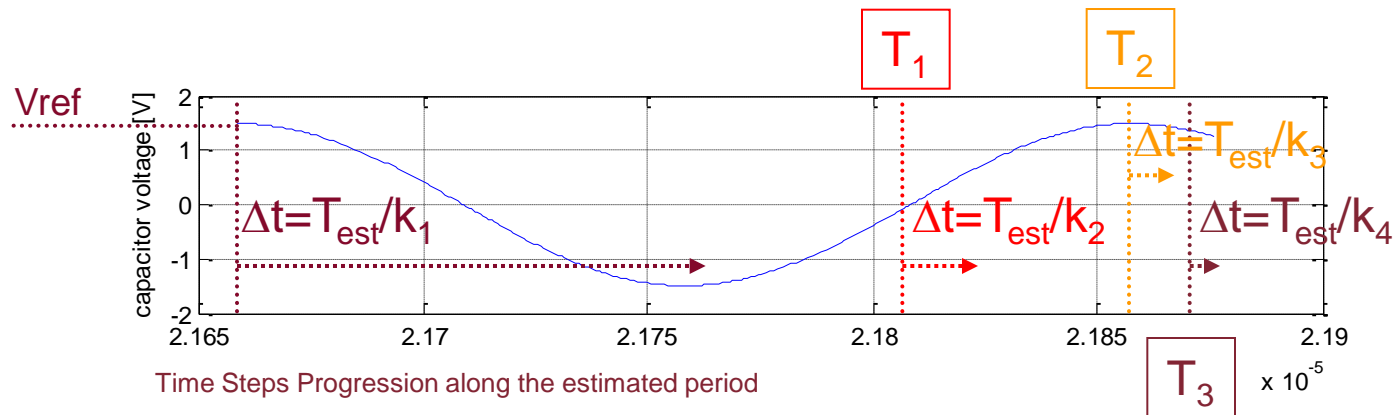
We can infer that RHS is a PSV, because it contains the solution due to equivalent conductance of the components with memory plus other stuff, like conductance of the components without memory, independent sources and controlled sources

It's true that a **state vector** should contain only state variables, but it can be demonstrated that a **pseudo-state vector** can be used to infer steady state condition

PSS Algorithm – Incremental Time Steps

During every accepted time point of transient evolution, the mean quadratic error of PSV in time and its derivative are computed. The calculated mean quadratic error, the PSV error and the derivative are used to update the period. This new period can be used in the next shooting iteration or can be the final one.

Transient time steps in the current estimated period are not constant:



The choice of $\{T_x\}$ and $\{k_x\}$ influences heavily:

- convergence accuracy
- algorithm speed
- convergence trend (monotonic, oscillating, etc.)

PSS Algorithm – Convergence Condition

Convergence criterion is based on the fulfillment of the following condition at every node or branch which satisfies a “dynamic consistency” test:

$$\begin{aligned} \text{voltage_error} &\leq (\max(V) * \text{rel_tol} + v_abstol) * \text{tr_tol} * \text{steady_coeff} \\ \text{current_error} &\leq (\max(I) * \text{rel_tol} + i_abstol) * \text{tr_tol} * \text{steady_coeff} \end{aligned}$$

The “dynamic consistency” test is based on the following condition:

$$\begin{aligned} \text{abs (RHS [max] – RHS [min])} &> 10\mu\text{V} \\ \text{abs (RHS [max] – RHS [min])} &> 10\text{nA} \end{aligned}$$

Convergence is reached when every node and branch verifies the condition

The following values are set as default in NGSPICE:

$$\begin{aligned} \text{rel_tol} &= 1\text{e-}3 \\ \text{v_abstol} &= 1\text{e-}6 \text{ V} \\ \text{i_abstol} &= 1\text{e-}9 \text{ A} \\ \text{tr_tol} &= 7 \end{aligned}$$

steady_coeff (grabbed from outside) ← inserted for PSS only

PSS Algorithm – PSS reached

When convergence is reached:

- circuit is evolved one more time at the final frequency with CONSTANT time steps
- results are stored for Time Domain plotting

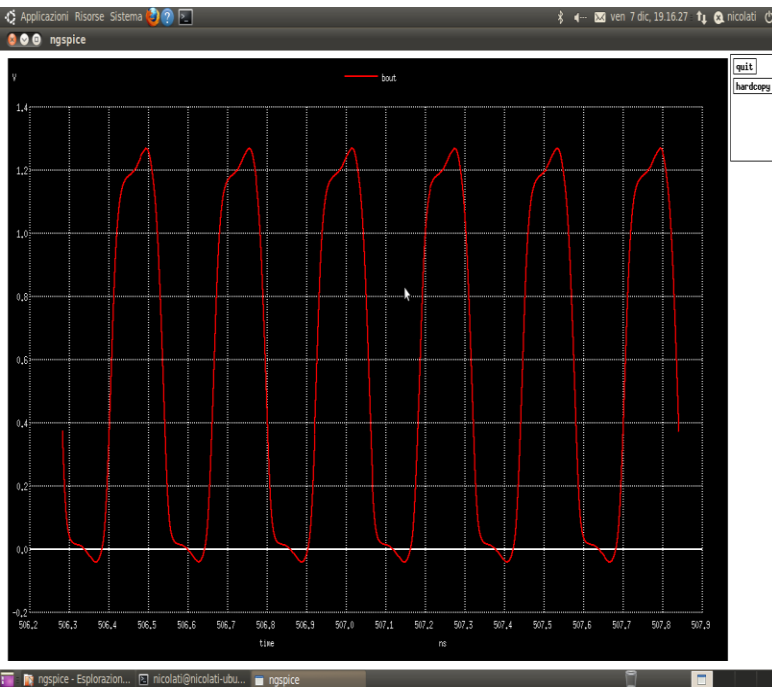


When all the Time Domain results are gathered, they are passed to a DFT routine that calculates the required harmonics and stores results for Frequency Domain plotting and Frequency Checking

PSS Algorithm – Frequency Check Trough DFT

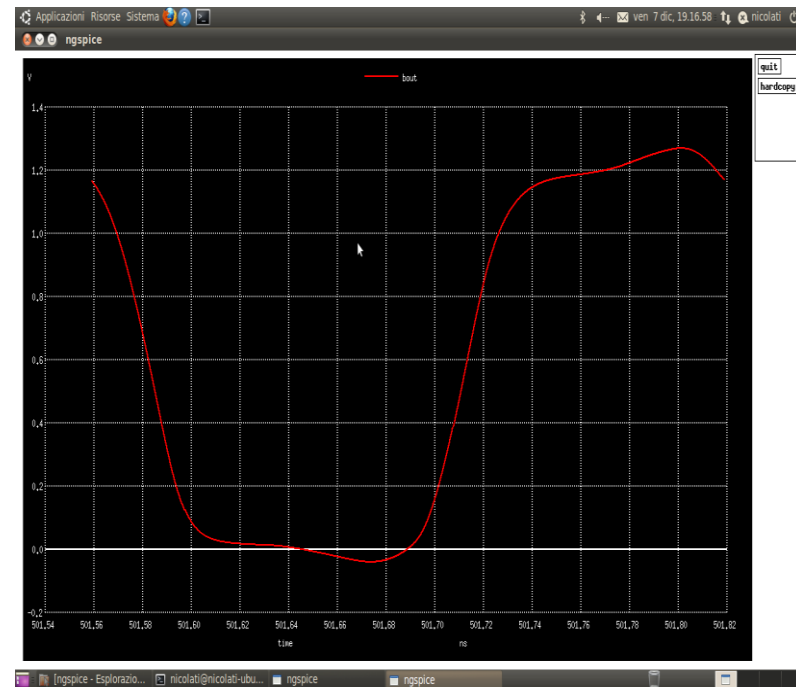
If the shooting cycle converges to a local minimum, expressed as multiple of the first harmonic, a check to estimate the correct frequency is performed through the data collected by the DFT

Generally, in this case, a very ‘far’ initial guessed frequency is provided: algorithm can lock into a local minimum for PSV error, without the chance to get out



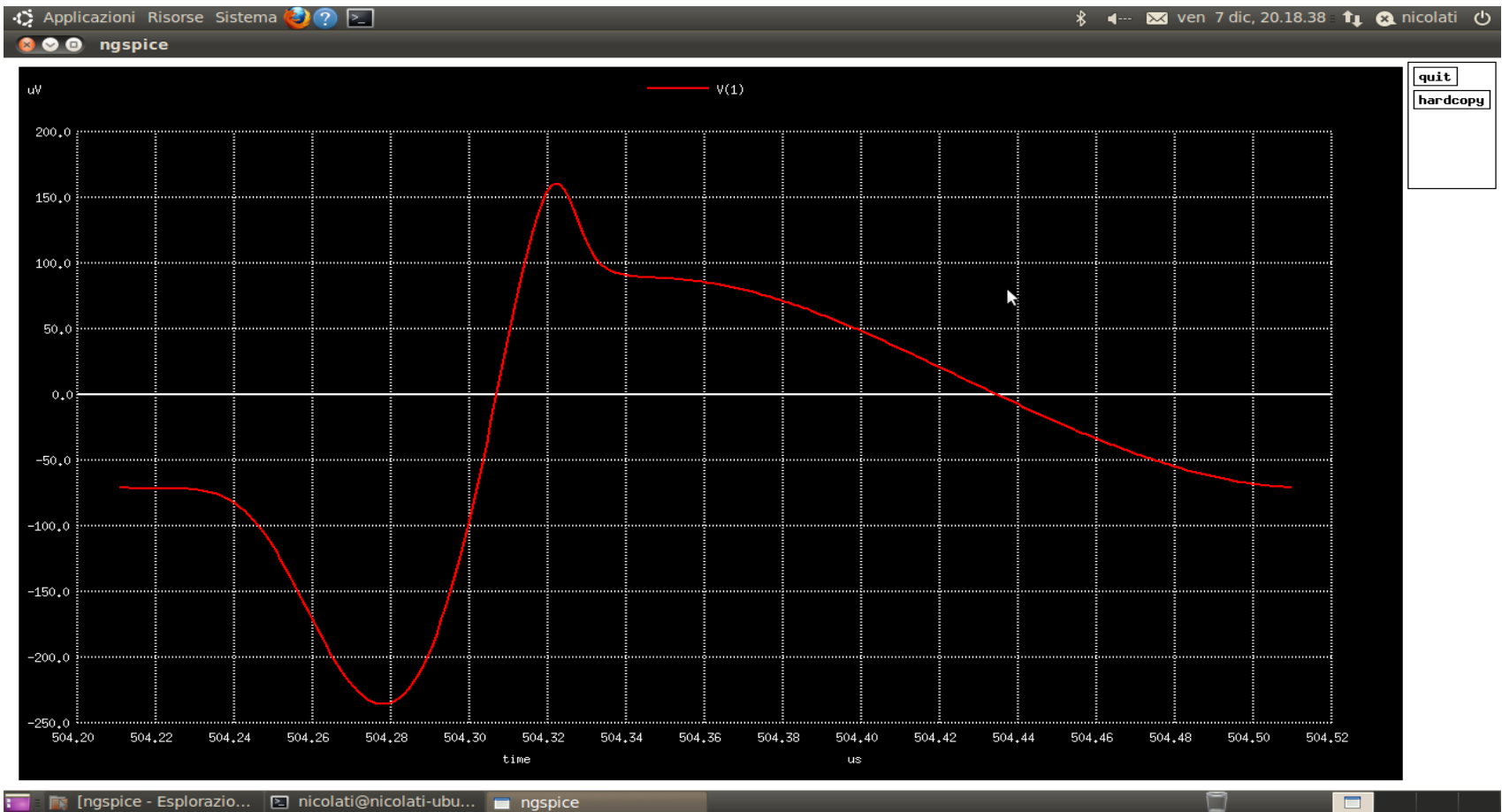
RING OSCILLATOR

Final circuit time is 5.062813106e-07 seconds (iteration n° 3) and predicted fundamental frequency is 641.4261187e+06 Hz. The predicted fundamental frequency is incorrect. Relaunching the analysis... The new guessed fundamental frequency is: 3.84856e+09. Convergence not reached. However the most near convergence iteration has predicted (iteration 5) a fundamental frequency of 3.848555145e+09 Hz



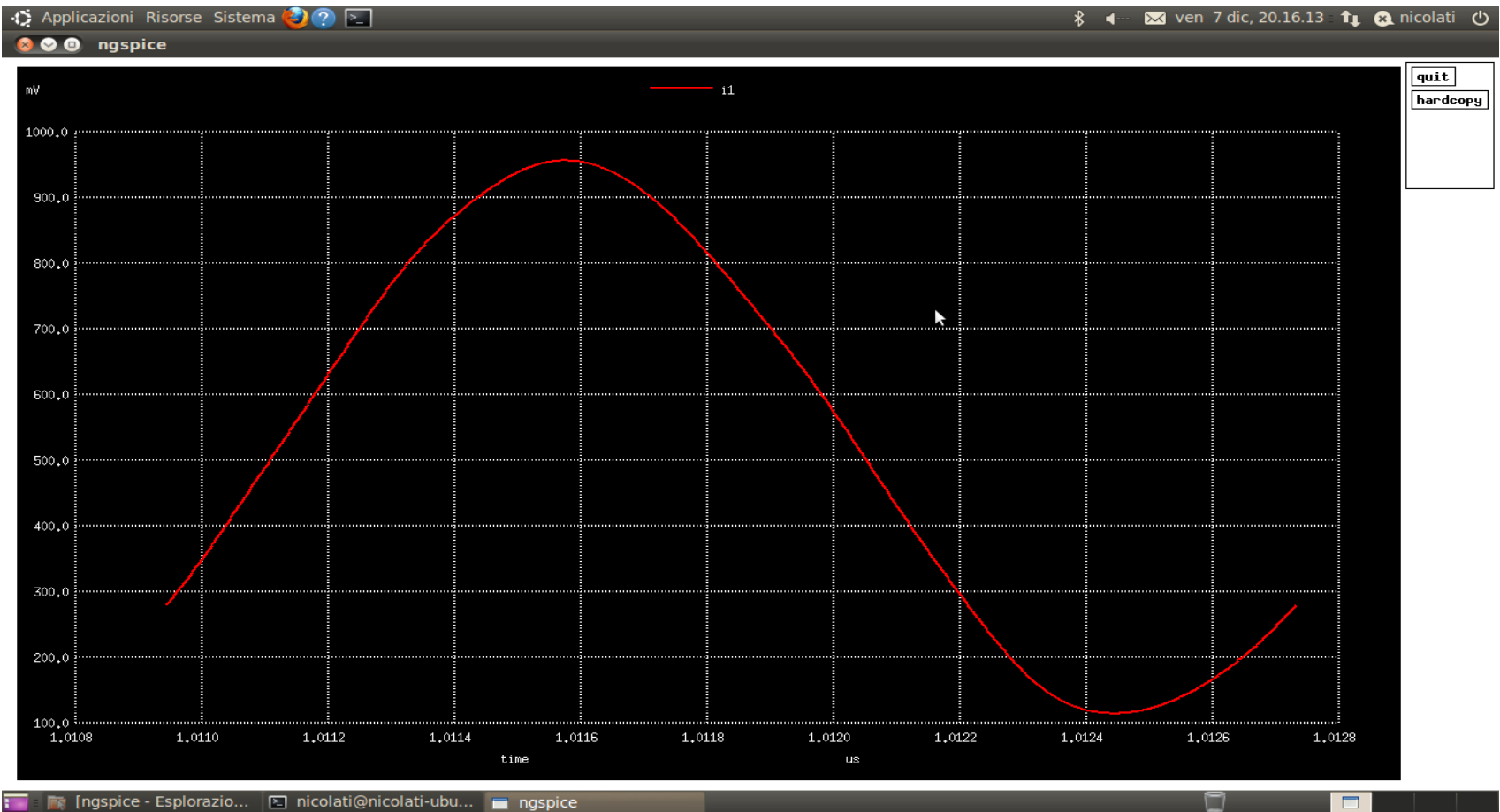
PSS Algorithm – Examples

Colpitt's Oscillator – $f=3.343\text{MHz}$



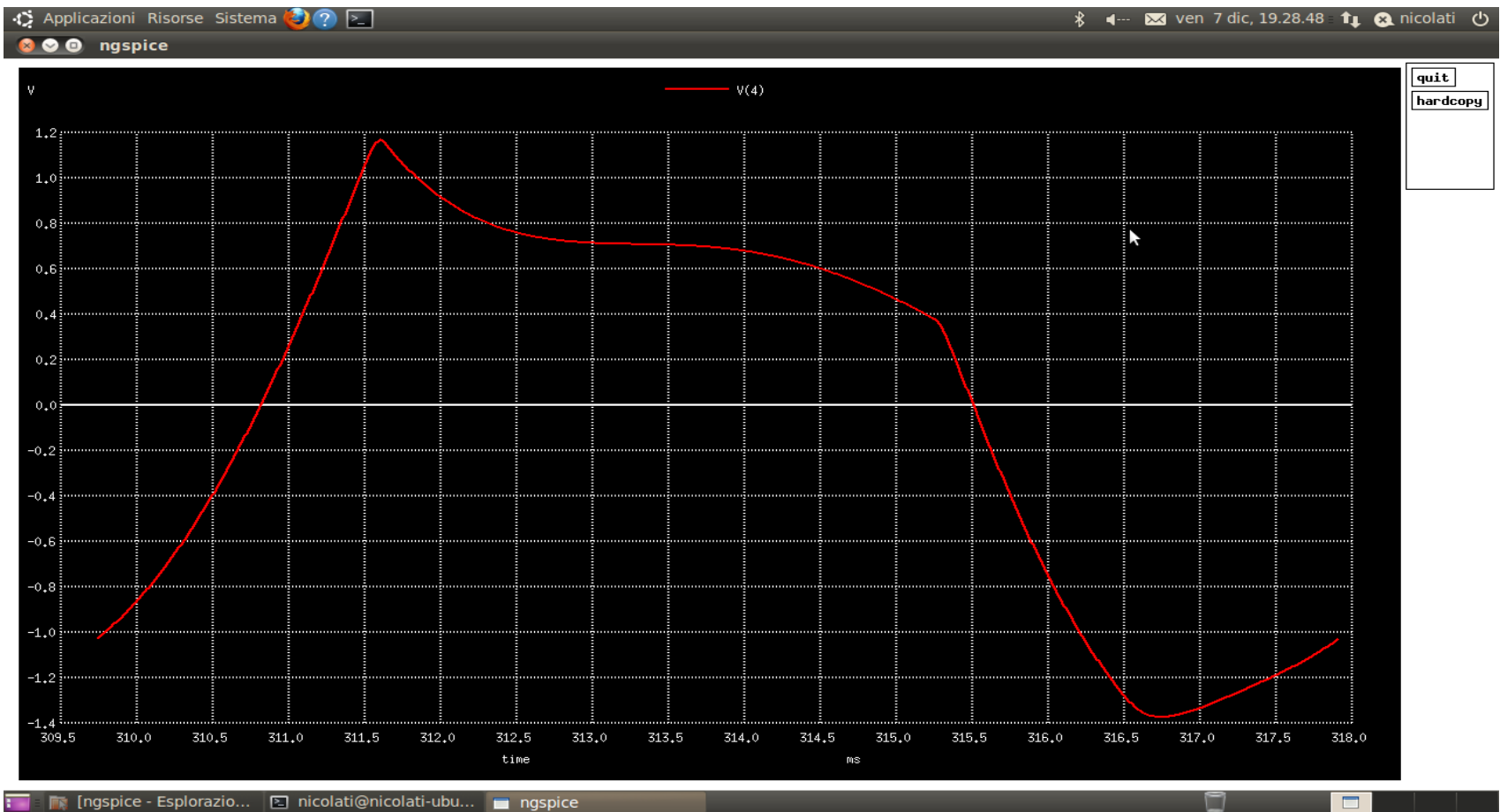
PSS Algorithm – Examples

Complementary CMOS Oscillator – $f=559.208\text{MHz}$



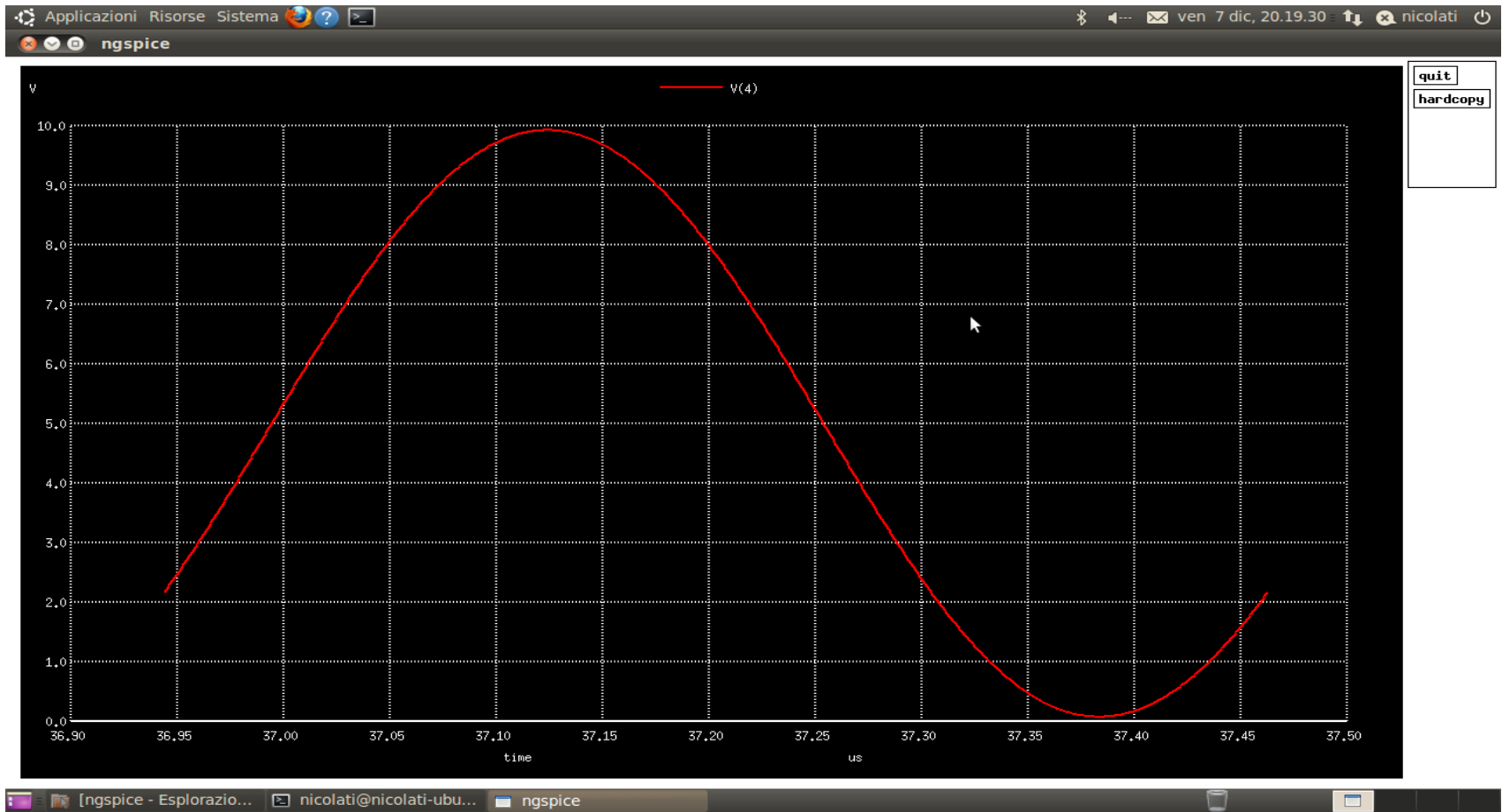
PSS Algorithm – Examples

Hartley Oscillator – $f=122.423\text{Hz}$



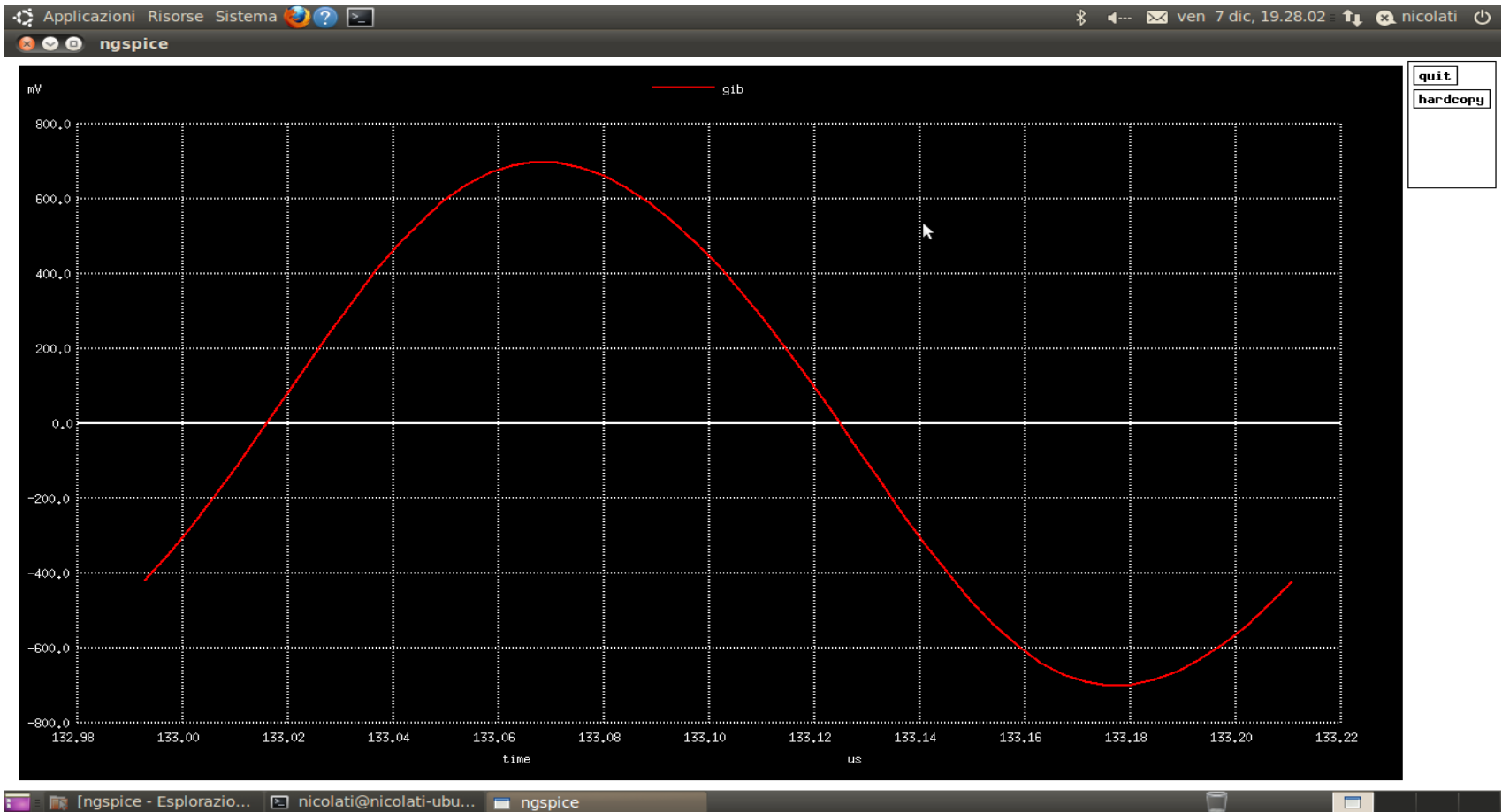
PSS Algorithm – Examples

Vackar Oscillator – $f=1.927\text{MHz}$



PSS Algorithm – Examples

Van Der Pol Oscillator – $f=4.590\text{MHz}$



Thanks for your attention



Appendix: KLU SPARSE DIRECT LINEAR SOLVER

Properties of the MNA matrices (Davis)

- Zero-free (or nearly zero-free) diagonal (zeroes removed by Maximum Transversal Algorithm)
- Unsymmetric values with a non-zero pattern only roughly symmetric
- Permutable to block triangular form (BTF)
- Non-zero pattern of each block is typically more symmetric than the whole matrix (peculiarity of circuit matrices)
- Presence of dense rows/columns (removed by COLAMD or AMD)
- Very sparse (using BLAS is not appropriate)
- Very sparse LU factors if properly ordered (peculiarity of circuit matrices)
- High number of fill-ins if ordered with usual strategies

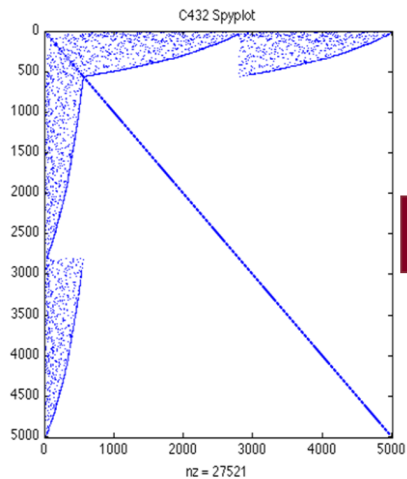
KLU exploits all these properties

Matrix Pre-Ordering in KLU

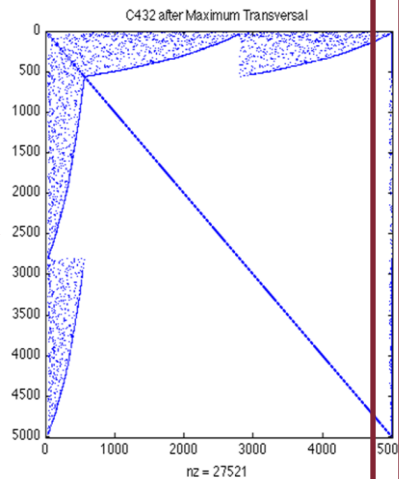
Pre-Ordering avoids zeroes along the main diagonal, reduces the workload and reduces the fill-ins

Pre-Ordering sequence performed on the transient simulation of the «c432» ISCAS85 suite circuit

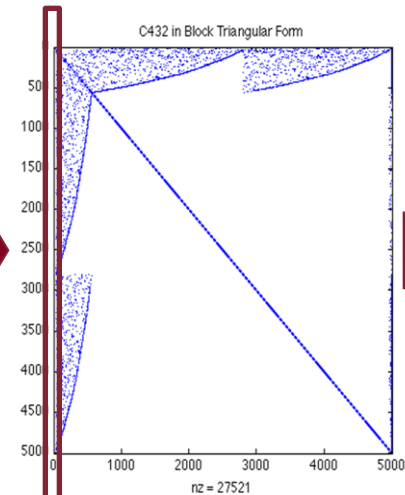
Original Sparse Matrix



Maximum Transversal

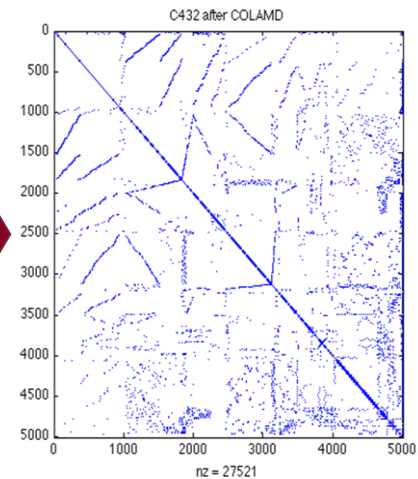


Block Triangular Form



Partitioning generates 40 blocks with a DOMINANT one.

COLAMD



Gilbert-Peierls LU Factorization in KLU

Dense matrix version

Base GP algorithm MATLAB pseudo code:

```
L = I
for each column j
  b = A(:, j)
  x = L \ b
  (Partial Pivoting)
  U(1:j, j) = x(1:j)
  L(j+1:n, j) = x(j+1:n) / U(j, j)
end
```

% To solve $x = L \setminus b$ the following algorithm is used:

```
x = b
for columns i in (1:j-1)
  x(i+1:n) = x(i+1:n) -
    L(i+1:n, i) * x(i)
end
```

Sparse matrix version

The sparse matrix version of the GP algorithm exploits the sparsity of L:

```
L = I
for each column j
  b = A(:, j)
  x = L \ b
  (Partial Pivoting)
  U(1:j, j) = x(1:j)
  L(j+1:n, j) = x(j+1:n) / U(j, j)
end
```

% To solve $x = L \setminus b$ the following algorithm is used:

```
x = b
for columns i in (1:j-1)
  if L(i+1:n, i) != 0
    x(i+1:n) = x(i+1:n) -
      L(i+1:n, i) * x(i)
  end
```

Triangular system with
Partial Pivoting

$$\begin{array}{l} x_1 \\ l_{21}x_1 + x_2 \\ l_{31}x_1 + l_{32}x_2 + l_{33}x_3 \\ l_{41}x_1 + l_{42}x_2 + l_{43}x_3 \\ \vdots \\ l_{n1}x_1 + l_{n2}x_2 + l_{n3}x_3 \end{array}$$

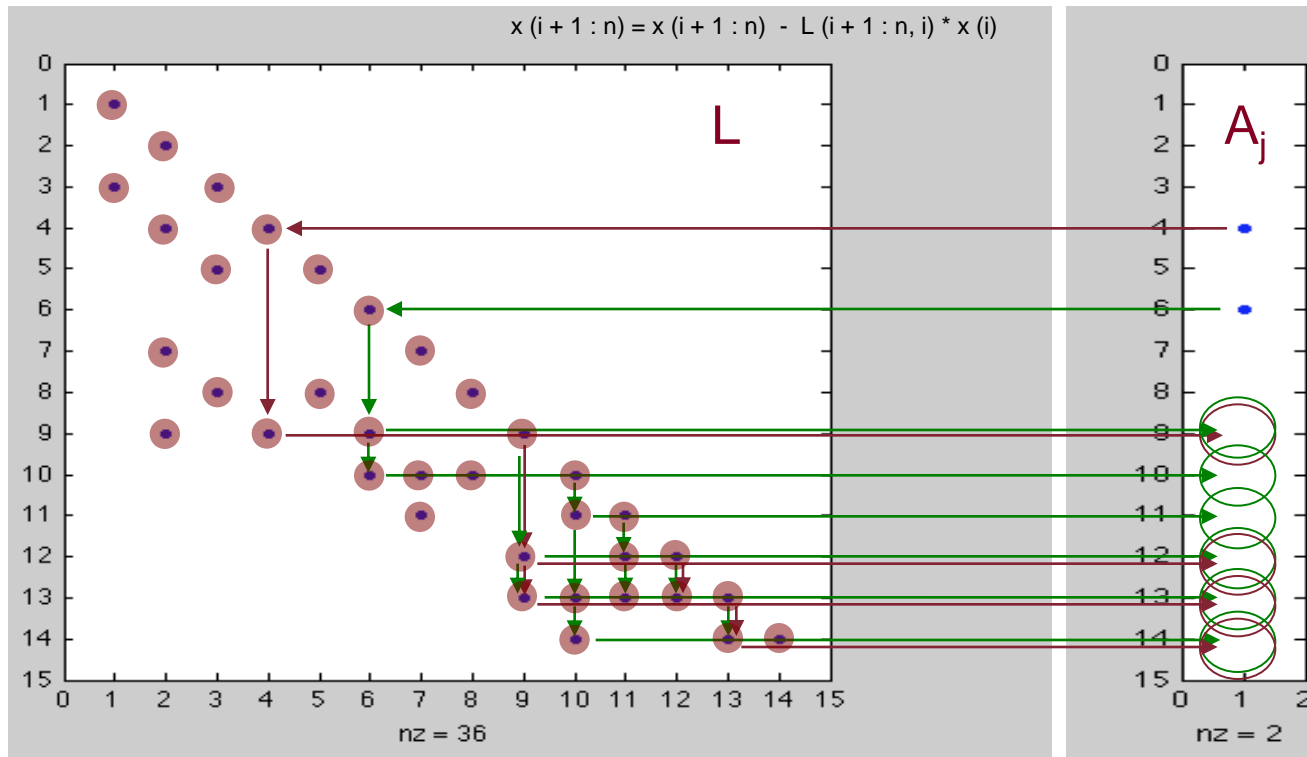
Triangular system
without Partial Pivoting

Pivot
Candidates

Gilbert-Peierls LU Factorization in KLU

Sparse matrix version improved through the DFS

How the DFS works in the Gilbert-Peierls algorithm



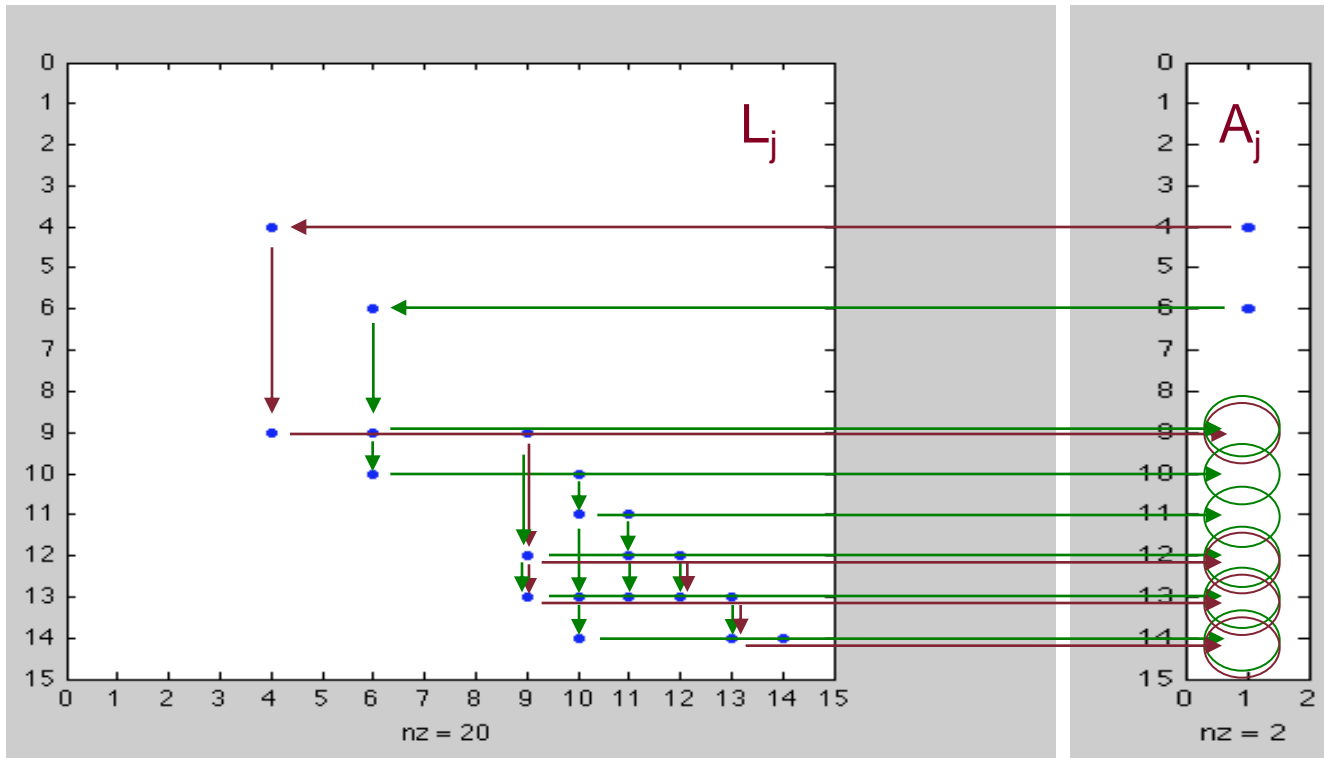
- This is a reachability problem
- The DFS (depth-first search) is applied on L, starting from each non-zero element of the j_{th} column of A

Considering all the non-zeros of L, it's necessary to access to 36 elements divided in 14 columns

Gilbert-Peierls LU Factorization in KLU

Reachability reduces the number of elements accessed

Reachability Set \rightarrow portion of L pointed by A_j



- Number of **elements** reduced by **44%**
- Number of **columns** reduced by **43%**

Considering L_j instead of L , it's necessary to access **only** to 20 elements in 8 columns

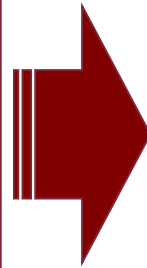
Gilbert-Peierls LU Factorization in KLU

Sparse Matrix version (as said before)

```
L = I
for each column j
  b = A (: , j)
  x = L \ b
  (Partial Pivoting)
  U (1 : j, j) = x (1 : j)
  L (j + 1 : n, j) = x (j + 1 : n) / U (j, j)
end

% To solve x = L \ b the following algorithm
is used:

x = b
for columns i in (1 : j - 1)
  if L (i + 1 : n, i) != 0
    x (i + 1 : n) = x (i + 1 : n) -
      L (i + 1 : n, i) * x (i)
  end
end
```



Improved Sparse Matrix version

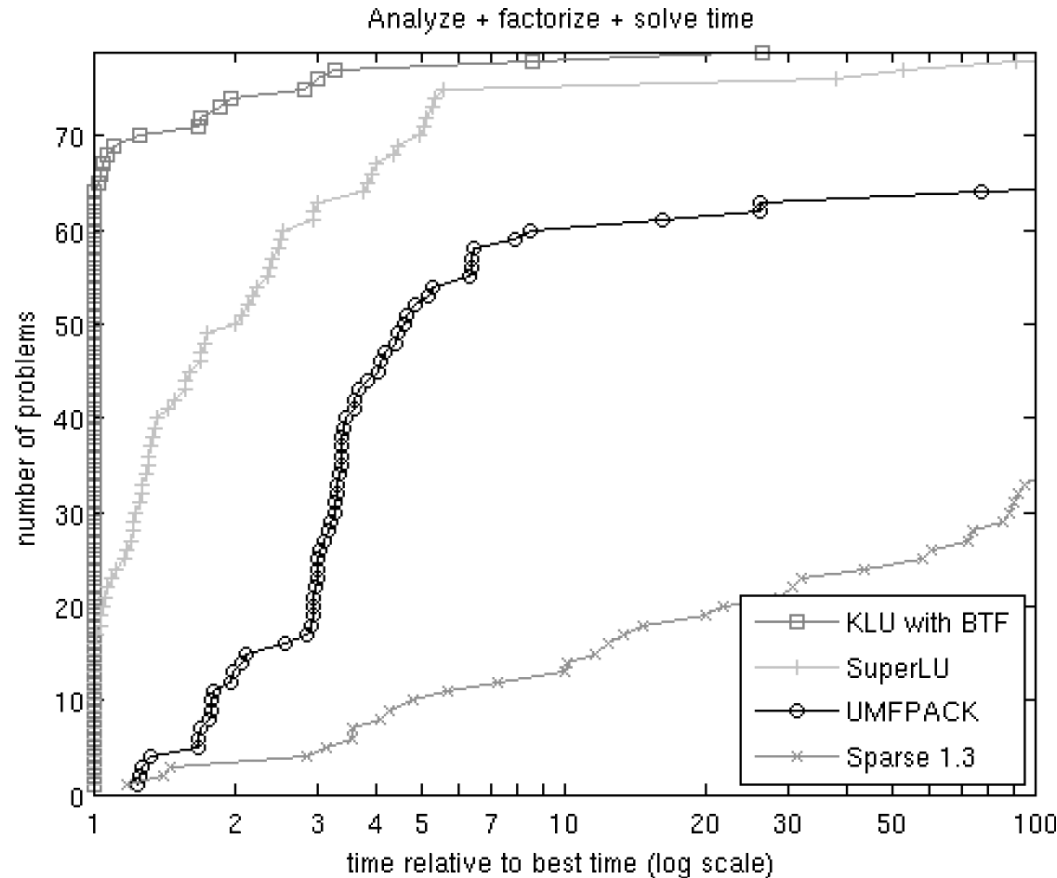
```
L = I
for each column j
  V = Reachability (L, A (: , j)) ←
  b = A (: , j)
  x = L \ b
  (Partial Pivoting)
  U (1 : j, j) = x (1 : j)
  L (j + 1 : n, j) = x (j + 1 : n) / U (j, j)
end

% To solve x = L \ b the following algorithm
is used :

x = b
for columns i in V ←
  if L (i + 1 : n, i) != 0
    x (i + 1 : n) = x (i + 1 : n) -
      L (i + 1 : n, i) * x (i)
  end
end
```

KLU Key Facts

- KLU exploits the structure of MNA matrices better than SPARSE 1.3 (pre-order)
- KLU minimizes the number of computations (GP Algorithm has a complexity $O(\text{flops}(\text{LU}))$)
- Features not shown:
 - **Symmetric pruning** to reduce DFS time
 - **Re-factorization**
 - DFS is not used during re-factorization



Reference: T. Davis, E. Natarajan, Algorithm 907: KLU, a direct sparse solver for circuit simulation problems, ACM Trans. on Math. Soft., 2010