
Subject: Re: Ideas: Grid lines at "round" dates or values. More options for tooltip.
Posted by [Didier](#) on Sat, 31 Aug 2019 10:49:29 GMT
[View Forum Message](#) <> [Reply to Message](#)

Hello Deep and Koldo,

It think I have exactly what you need but it is not directly applicable to ScatterCtrl (I use it with my own plotting ctrl which has different grid/axis management).

- * it does exactly what you want with ordinary numeric grids
- * it does the it's best whith dates (or at least what seemed good for my needs :).
since months don't have the same number of days : 29, 30 or 31, it get's quite difficult to have something good looking)
- * it also manages LOG grid steps
- * any custom grid calculation can be plugged in using the `setGridStepCalcBack()` method

NOTE : the nbr of grid steps takes in account the screen size of the plot

I use dedicated classes to manage the grid steps

The strategy is the following:

- * when zoom/scroll/resize ==> recompute the grid which sets an array (`_stepDrawingParams`) containing all the info for the calculated grid steps (position, text to be displayed, and custom data)
- * when drawing needed : the array is used to get the information on the steps

What you are searching for is int the following methods:

```
void stdGridStepCalcCbk(CLASSNAME& gridStepManager, CoordinateConverter& coordConv
);
void log10GridStepCalcCbk(CLASSNAME& gridStepManager, CoordinateConverter&
coordConv );
void dateGridStepCalcCbk(CLASSNAME& gridStepManager, CoordinateConverter& coordConv
);
void timeGridStepCalcCbk(CLASSNAME& gridStepManager, CoordinateConverter& coordConv
);
```

Here is the class I use for this.

GridStepManager.h

```
#ifndef _GraphCtrl_GridStepManager_h_
#define _GraphCtrl_GridStepManager_h_

#include <limits>
```

```

namespace GraphDraw_ns
{

// =====
//  GridStepManager  CLASS
// =====
class GridStepData {
public:
    unsigned int tickLevel; // 0:Major tick  1:secondary tick  2: ....
    bool drawTickText;
    Size_<Upp::uint16>  textSize;
    TypeGraphCoord stepGraphValue;
    String      text;
    Value      customParams; // general purpose step parameter holder : filled when calculating
the steps

    GridStepData()
    : tickLevel(0)
    , drawTickText(true)
    {
    }
};

class GridStepIterator {
private:
    int _nbSteps; // current step range (according to grid step)
    int _currentStep; // current step number
    CoordinateConverter& _coordConverter;
    GridStepData* const _stepData; // points to steps parameters array (pre-calculated by
GridStepManager)

public:
    typedef GridStepIterator CLASSNAME;

    GridStepIterator( CoordinateConverter& conv, int nbSteps, GridStepData* stepData, int
currentStep)
    : _nbSteps(nbSteps)
    , _currentStep(currentStep)
    , _coordConverter(conv)
    , _stepData(stepData)
    {}

    GridStepIterator(const CLASSNAME& p)
    : _nbSteps(p._nbSteps)
    , _currentStep(p._currentStep)
    , _coordConverter(p._coordConverter)
    , _stepData(p._stepData)

```

```

{}

public:
inline bool operator==( const CLASSNAME& v) const { return( _currentStep==v._currentStep ); }
inline bool operator!=( const CLASSNAME& v) const { return( _currentStep!=v._currentStep ); }
inline operator TypeScreenCoord() const { return _coordConverter.toScreen( getGraphValue() ); }
}
inline operator Size() const { return Size(_stepData[_currentStep].textSize.cx,
_stepData[_currentStep].textSize.cy); }
inline operator String() const { return _stepData[_currentStep].text; }
GridStepData* operator->() { return _stepData + _currentStep; }

inline TypeGraphCoord getGraphValue() const { return
_stepData[_currentStep].stepGraphValue; }
inline const Value& getCustomStepData() const { return
_stepData[_currentStep].customParams; }
inline unsigned int getTickLevel() const { return _stepData[_currentStep].tickLevel; }
inline bool drawTickText() const { return _stepData[_currentStep].drawTickText; }
inline TypeGraphCoord getGraphRange() const { return
_coordConverter.getSignedGraphRange(); }
inline int getStepNum() const { return _currentStep; }
inline int getNbSteps() const { return _nbSteps; }
inline bool isFirst() const { return (_currentStep==0); }
inline bool isLast() const { return (_currentStep==( _nbSteps-1)); }

inline CLASSNAME& toEnd() { _currentStep = _nbSteps; return *this; }
inline CLASSNAME& toBegin() { _currentStep = 0; return *this; }

// ++X
inline CLASSNAME& operator++()
{
++_currentStep;
return *this;
}

// X++
CLASSNAME operator++(int)
{
CLASSNAME tmp(*this);
++_currentStep;
return tmp;
}
};

typedef Callback2< const GridStepIterator&, String&> TypeFormatTextCb; // IN: valueIterator,
OUT: formatted value

```

```

class GridStepManager
{
public:
    enum { NB_MAX_STEPS = 100 };
    typedef GridStepManager CLASSNAME;
    typedef GridStepIterator Iterator;
    typedef Callback2<CLASSNAME&, CoordinateConverter&> TypeGridStepCalcCallBack;

protected:
    double _textSize; // tick Text MAX Size ( can be height or width )
    unsigned int _nbMaxSteps; // steps range is : [0, _nbMaxSteps]
    unsigned int _nbSteps; // current step range (according to grid step)
    unsigned int _currentStep; // current step number
    CoordinateConverter& _coordConverter;
    bool updateNeeded;

    GridStepData _stepDrawingParams[NB_MAX_STEPS+1]; // general purpose step parameter
    holder : filled when calculating the steps

    TypeGridStepCalcCallBack _updateCb;

    ChangeStatus coordConvLastStatus;

    TypeGraphCoord GetNormalizedStep(TypeGraphCoord range) const
    {
        int e = 0;
        double s = Upp::normalize(range/(_nbMaxSteps+1), e);
        double sign = 1;
        if (s<0) sign = -1;

        s = sign*s;
        if ( (s>2) && (s<=5) ) { s = 5; }
        else if( (s>1) && (s<=2) ) { s = 2; }
        else if( (s>5) && (s<=10) ) { s = 1; e++; }
        else { s = 1; }
        return sign * s * Upp::ipow10(e);
    }

    template <class T>
    T GetNormalizedStep(TypeGraphCoord range, const Vector<T>& stepValues) const
    {
        double s = range/(_nbMaxSteps+1);
        T sign = 1;
        if (s<0) sign = -1;

        s = sign*s;
        int c=0;
        while (c < stepValues.GetCount()-1) {

```

```

    if ( (s>stepValues[c]) && (s<=stepValues[c+1]) ) {
        s = stepValues[c+1];
        return sign*s;
    }
    ++c;
}
if (c==stepValues.GetCount()) s = stepValues[0];
return sign*s;
}

```

TypeGraphCoord GetGridStartValue(TypeGraphCoord pstepValue, TypeGraphCoord pgraphMin) const

```

{
    TypeGraphCoord res;
    long double stepValue = pstepValue;
    long double graphMin = pgraphMin;
    if (graphMin>=0) {
        res = ((Upp::int64)(graphMin/stepValue + 1.0 -
std::numeric_limits<TypeGraphCoord>::epsilon())) * stepValue;
    } else {
        res = (((Upp::int64)(graphMin/stepValue) ) * stepValue);
    }
    if (res < pgraphMin) res = pgraphMin;
    return res;
}

```

```

void stdGridStepCalcCbk(CLASSNAME& gridStepManager, CoordinateConverter& coordConv
);
void log10GridStepCalcCbk(CLASSNAME& gridStepManager, CoordinateConverter&
coordConv );
void dateGridStepCalcCbk(CLASSNAME& gridStepManager, CoordinateConverter& coordConv
);
void timeGridStepCalcCbk(CLASSNAME& gridStepManager, CoordinateConverter& coordConv
);

```

```

public:
    GridStepManager(CoordinateConverter& coordConv)
    : _textSize(30)
    , _nbMaxSteps( Upp::min(15, (int)NB_MAX_STEPS) )
    , _currentStep( 0 )
    , _coordConverter( coordConv )
    {
        setStdGridSteps();
        UpdateGridSteps();
    }

```

```

GridStepManager(int nbSteps, int currStep, CoordinateConverter& coordConv)
: _textSize(30)

```

```

, _nbMaxSteps( Upp::max(5, Upp::min(nbSteps, (int)NB_MAX_STEPS)) )
, _currentStep( currStep )
, _coordConverter( coordConv )
{
    setStdGridSteps();
    UpdateGridSteps();
}

private:
// explicitly forbidden to use
CLASSNAME& operator=( const CLASSNAME& v) { return *this; }
//GridStepManager()
//: _textSize(30)
//, _nbMaxSteps( 0)//CLASSNAME::NB_MAX_STEPS) )
//, _currentStep( 0 )
//, _coordConverter()
//{}

public:
void setGridStepCalcBack(TypeGridStepCalcCallBack cbk) { _updateCbk = cbk; updateNeeded
= true; }
void setStdGridSteps() { _updateCbk = THISBACK(stdGridStepCalcCbK);
updateNeeded = true; }
void setLogGridSteps() { _updateCbk =
THISBACK(log10GridStepCalcCbK); updateNeeded = true; }
void setTimeGridSteps() { _updateCbk = THISBACK(timeGridStepCalcCbK);
updateNeeded = true; }
void setDateGridSteps() { _updateCbk = THISBACK(dateGridStepCalcCbK);
updateNeeded = true; }

virtual ~GridStepManager() {}

virtual void UpdateGridSteps()
{
    //RLOGBLOCK("UpdateGridSteps()");
    if ( updateNeeded || coordConvLastStatus.hasChangedUpdate(_coordConverter) ) {
        if ( _textSize > 1.0) _nbMaxSteps = _coordConverter.getScreenRange()/_textSize;
        _updateCbk(*this, _coordConverter);
        updateNeeded = false;
    }
}

void SetNbSteps(int nbSteps)
{
    //_nbMaxSteps = nbSteps;
}

```

```

int GetNbSteps() const { return _nbSteps; }

bool SetTextMaxSize(double textSize)
{
    textSize *= 2;
    if ( _textSize < textSize) {
        if ( textSize/_textSize > 1.2) {
            //RLOG("> SetTextMaxSize(>1.2)(" << textSize << ") old _textSize=" << _textSize << "
---UPDATED---");
            _textSize = textSize;
            updateNeeded = true;
            return true;
        }
    } else {
        if (textSize/_textSize < 0.5) {
            //RLOG("> SetTextMaxSize(<0.5)(" << textSize << ") old _textSize=" << _textSize << "
---UPDATED---");
            _textSize = textSize;
            updateNeeded = true;
            return true;
        }
    }
    return false;
}

Iterator End() {
    return GridStepIterator( _coordConverter, _nbSteps+1, _stepDrawingParams, _nbSteps+1);
}

Iterator Begin() {
    return GridStepIterator( _coordConverter, _nbSteps+1, _stepDrawingParams, 0);
}

};

};
#endif

```

```

GridStepManager.cpp
#include "GraphDraw.h"

```

```

#define GRIDLOG(a) // LOG(a)

```

```

NAMESPACE_UPP
namespace GraphDraw_ns

```

```

{
void GridStepManager::stdGridStepCalcCbk(GridStepManager& gridStepManager,
CoordinateConverter& coordConv )
{
    TypeGraphCoord gridStepValue = GetNormalizedStep(
(float)coordConv.getSignedGraphRange() ); // DO NOT REMOVE (float) ==> prevents artifacts du
to range precision errors when scrolling (ex: range changes from 14.000000000000052 to
13.999999999999997)
    TypeGraphCoord gridStartValue = GetGridStartValue( gridStepValue, coordConv.getGraphMin()
);
    _nbSteps = (unsigned int)tabs((coordConv.getGraphMax() - gridStartValue) / gridStepValue);

    if (_nbSteps > _nbMaxSteps) {
        _nbSteps = _nbMaxSteps;
    }

    GRIDLOG("stdGrid [" << (int) this << "] =====");
    // fill step values ==> used by gridStepIterator
    for (unsigned int c=0; c<_nbSteps+1; ++c) {
        _stepDrawingParams[c].stepGraphValue = gridStartValue + gridStepValue*c;
        _stepDrawingParams[c].drawTickText = 1; // ==> draw tick text
        GRIDLOG("stdGrid [" << (int) this << "] - step[" << c << "] = " <<
_stepDrawingParams[c].stepGraphValue );
    }
    GRIDLOG("stdGrid [" << (int) this << "] SignedRange = " <<
coordConv.getSignedGraphRange() );
    GRIDLOG("stdGrid [" << (int) this << "] getGraphMin = " << coordConv.getGraphMin() );
    GRIDLOG("stdGrid [" << (int) this << "] getGraphMax = " << coordConv.getGraphMax() );
    GRIDLOG("stdGrid [" << (int) this << "] gridStartValue=" << gridStartValue );
    GRIDLOG("stdGrid [" << (int) this << "] gridStepValue = " << gridStepValue );
    GRIDLOG("stdGrid [" << (int) this << "] screenRange = " << coordConv.getScreenRange() );
    GRIDLOG("stdGrid [" << (int) this << "] screenMin = " << coordConv.getScreenMin() );
    GRIDLOG("stdGrid [" << (int) this << "] screenMax = " << coordConv.getScreenMax() );
    GRIDLOG("stdGrid [" << (int) this << "] _nbSteps = " << _nbSteps );
    GRIDLOG("stdGrid [" << (int) this << "] _nbMaxSteps = " << _nbMaxSteps );
    GRIDLOG("stdGrid [" << (int) this << "] _textSize = " << _textSize );

}

void GridStepManager::log10GridStepCalcCbk(GridStepManager& gridStepManager,
CoordinateConverter& coordConv )
{
    const double rangeFactor = coordConv.getGraphMax()/coordConv.getGraphMin();
    if ( rangeFactor < 10 ) {
        stdGridStepCalcCbk(gridStepManager, coordConv);
    }
    else

```



```

{
    const double logRangeFactor = floor(log10(rangeFactor));
    int logStepValue = GetNormalizedStep( (float)logRangeFactor );
    if (logStepValue == 0) logStepValue = 1;

    TypeGraphCoord gridStartValue = GetGridStartValue(logStepValue,
log10(coordConv.getGraphMin()));
    --gridStartValue;
    unsigned int nbLogSteps = (log10(coordConv.getGraphMax())-gridStartValue) / logStepValue;
    if (nbLogSteps > _nbMaxSteps) {
        nbLogSteps = _nbMaxSteps;
    }

    double logStepDensity = 2;
    if ( nbLogSteps > 0 ) logStepDensity = (_nbMaxSteps*logStepValue)/logRangeFactor;

    GRIDLOG("=====");
    GRIDLOG("log10 getGraphMin  =" << coordConv.getGraphMin() );
    GRIDLOG("log10 getGraphMax  =" << coordConv.getGraphMax() );
    GRIDLOG("log10 rangeFactor  =" << rangeFactor );
    GRIDLOG("log10 logRangeFactor=" << logRangeFactor );
    GRIDLOG("log10 logStepValue  =" << double(logStepValue) );
    GRIDLOG("log10 gridStartValue=" << gridStartValue );
    GRIDLOG("log10 _nbMaxSteps   =" << _nbMaxSteps );
    GRIDLOG("log10 nbLogSteps    =" << nbLogSteps );
    GRIDLOG("log10 logStepDensity=" << logStepDensity );
    GRIDLOG("log10 _textSize     =" << _textSize );

    // fill step values ==> used by gridSteplerator
    _nbSteps = -1;
    for (unsigned int c=0; c<(nbLogSteps+1); ++c)
    {
        double stepValue = pow( 10, gridStartValue + logStepValue*c );
        double currVal = stepValue;

        if ( (currVal > coordConv.getGraphMin()) && (currVal < coordConv.getGraphMax())) {
            ++_nbSteps;
            _stepDrawingParams[_nbSteps].drawTickText = 1; // ==> draw tick text
            _stepDrawingParams[_nbSteps].tickLevel = 0;
            _stepDrawingParams[_nbSteps].stepGraphValue = currVal;
        }
        currVal += stepValue;

        typedef struct {
            bool isStep[9];
            bool displayText[9];
        } Log10Steps;

```

```

const Log10Steps* logSteps = 0;
if (logStepDensity < 2. || logStepValue>1)
{}
else if (logStepDensity < 3.) {
    static const Log10Steps tmpLogSteps = {
        //1 2 3 4 5 6 7 8 9
        { 1,0,1,0,1,0,1,0,1},
        { 1,0,0,0,1,0,0,0,0}
    };
    logSteps = &tmpLogSteps;
} else if (logStepDensity < 5.) {
    static const Log10Steps tmpLogSteps = {
        //1 2 3 4 5 6 7 8 9
        { 1,1,1,1,1,1,1,1,1},
        { 1,1,0,0,1,0,0,0,0}
    };
    logSteps = &tmpLogSteps;
} else if (logStepDensity < 7.) {
    static const Log10Steps tmpLogSteps = {
        //1 2 3 4 5 6 7 8 9
        { 1,1,1,1,1,1,1,1,1},
        { 1,1,1,0,1,0,0,0,0}
    };
    logSteps = &tmpLogSteps;
} else if (logStepDensity < 11.) {
    static const Log10Steps tmpLogSteps = {
        //1 2 3 4 5 6 7 8 9
        { 1,1,1,1,1,1,1,1,1},
        { 1,1,1,1,1,0,1,0,0}
    };
    logSteps = &tmpLogSteps;
}
else
{
    static const Log10Steps tmpLogSteps = {
        //1 2 3 4 5 6 7 8 9
        { 1,1,1,1,1,1,1,1,1},
        { 1,1,1,1,1,1,1,1,1}
    };
    logSteps = &tmpLogSteps;
}
// fill log steps according to desired pattern configured above
if (logSteps) {
    for (unsigned int l=1; l < 9; ++l)
    {
        if (logSteps->isStep[l]) {
            if ( (currVal > coordConv.getGraphMin()) && (currVal < coordConv.getGraphMax())) {
                ++_nbSteps;
            }
        }
    }
}

```

```

        _stepDrawingParams[_nbSteps].drawTickText = logSteps->displayText[l];
        _stepDrawingParams[_nbSteps].stepGraphValue = currVal;
        _stepDrawingParams[_nbSteps].tickLevel = 1-logSteps->displayText[l];
    }
}
currVal += stepValue;
}
}
GRIDLOG("log10 step[" << c << "] = " << _stepDrawingParams[c].stepGraphValue << "
nbLogSteps="<<nbLogSteps);
}
}
}

void GridStepManager::dateGridStepCalcCbk(GridStepManager& gridStepManager,
CoordinateConverter& coordConv )
{
    Date dateRange;    dateRange.Set(coordConv.getSignedGraphRange());
    Date graphStartDate; graphStartDate.Set(coordConv.getGraphMin());
    Date graphEndDate;  graphEndDate.Set(coordConv.getGraphMin());
    // =====
    //     YEARS
    // =====
    if (dateRange.year >= 7) {
        int yearRange = dateRange.year;
        int normalizedYearStep = GetNormalizedStep( yearRange );
        if (normalizedYearStep==0) normalizedYearStep = 1;

        Date datelater(0,1,1);
        datelater.year = GetGridStartValue( normalizedYearStep, graphStartDate.year+1 );
        if ( datelater < graphStartDate ) {
            GRIDLOG(" ##### YEAR STEP_2 :  datelater="<< datelater << "    graphStartDate="<<
graphStartDate);
            datelater.year += normalizedYearStep;
        }

        _nbSteps = (unsigned int)tabs((graphEndDate.year - datelater.year) / normalizedYearStep);
        if (_nbSteps > _nbMaxSteps) {
            _nbSteps = _nbMaxSteps;
        }
        else if (_nbSteps==0) _nbSteps = 1;

        GRIDLOG("YEAR  range=" << yearRange << "years    normalizedStep = "<<
normalizedYearStep << " years    _nbSteps=" << _nbSteps<< "    graphStartDate = "<<
datelater);
        for (unsigned int c=0; c<= _nbSteps; ++c) {
            Time tmp = ToTime(datelater);
            _stepDrawingParams[c].stepGraphValue = tmp.Get();

```

```

GRIDLOG("  YEAR STEP :  datelter=" << datelter);
datelter = AddYears(datelter, normalizedYearStep);
}
}
// =====
//  MONTHS
// =====
else if ((dateRange.month >= 7) || (dateRange.year > 0 )) {
    int monthRange = dateRange.year*12 + dateRange.month;
    Vector<double> monthSteps;
    monthSteps << 1 << 2 << 3 << 4 << 6 << 12 << 24 << 48;
    int normalizedMonthStep = GetNormalizedStep( monthRange, monthSteps );
    if (normalizedMonthStep==0) normalizedMonthStep = 1;

    Date datelter(0,1,1);

    int nbMonths = GetGridStartValue( normalizedMonthStep,
graphStartDate.year*12+graphStartDate.month-1 );
    datelter.year = nbMonths/12;
    datelter.month= nbMonths - datelter.year*12 + 1;
    datelter.day = 1;
    GRIDLOG("  MONTH STEP_1 : " << datelter << "      nbMonths=" << nbMonths << "
graphStartDate=" << graphStartDate);
    if ( datelter < graphStartDate ) {
        GRIDLOG("  ##### MONTH STEP_2 :  datelter=" << datelter << "      graphStartDate=" <<
graphStartDate);
        datelter = AddMonths(datelter, normalizedMonthStep);
    }

    _nbSteps = (unsigned int)tabs(((graphEndDate.year-datelter.year)*12 +
(graphEndDate.month-datelter.month)) / normalizedMonthStep);
    if (_nbSteps > _nbMaxSteps) {
        _nbSteps = _nbMaxSteps;
    }
    else if (_nbSteps==0) _nbSteps = 1;

    GRIDLOG("MONTH range=" << monthRange << " months  normalizedMonthStep = " <<
normalizedMonthStep << " months  _nbSteps=" << _nbSteps << "      graphStartDate = " <<
graphStartDate);
    for (unsigned int c=0; c<= _nbSteps; ++c) {
        Time tmp = ToTime(datelter);
        _stepDrawingParams[c].stepGraphValue = tmp.Get();
        GRIDLOG("  MONTH STEP : " << datelter);
        datelter = AddMonths(datelter, normalizedMonthStep);
    }
}
// =====
//  DAYS

```

```

// =====
else {
    TypeGraphCoord gridStartValue;
    Upp::int64 normalizedStep;
    Vector<double> daySteps;
    daySteps << 1 << 2 << 3 << 4 << 5 << 7 << 14 << 28 << 56;
    normalizedStep = GetNormalizedStep( dateRange.Get(), daySteps );
    if (normalizedStep == 0) normalizedStep = 1;
    gridStartValue = GetGridStartValue( normalizedStep, _coordConverter.getGraphMin() );
    _nbSteps = (unsigned int)tabs((graphEndDate.Get() - gridStartValue) / normalizedStep);

    if (_nbSteps > _nbMaxSteps) {
        _nbSteps = _nbMaxSteps;
    }
    else if (_nbSteps==0) _nbSteps = 1;

    GRIDLOG("D/H/M/S ==> _nbSteps=" << _nbSteps << "    gridStepValue=" <<
normalizedStep);
    for (unsigned int c=0; c<_nbSteps+1; ++c)
    {
        _stepDrawingParams[c].stepGraphValue = gridStartValue + normalizedStep * c;
    }
}

void GridStepManager::timeGridStepCalcCbk(GridStepManager& gridStepManager,
CoordinateConverter& coordConv )
{
    enum {
        DAY_sec    = 24*60*60,
        HOUR_sec   = 60*60,
        MINUTES_sec = 60,
    };
    Time timeOrigin;    timeOrigin.Set(0);
    Time graphStartTime; graphStartTime.Set(coordConv.getGraphMin());
    Time graphEndTime;  graphEndTime.Set(coordConv.getGraphMax());
    Time timeRange;     timeRange.Set(coordConv.getSignedGraphRange());

    GRIDLOG("\n--timeGridStepCalcCbk--  range=" << timeRange << "    GraphMin = " <<
graphStartTime << "    GraphMax = " << graphEndTime);

    // =====
    //    YEARS
    // =====
    if (timeRange.year >= 7) {
        Date graphStartDate( graphStartTime.year, graphStartTime.month, graphStartTime.day);
        Date graphEndDate( graphEndTime.year, graphEndTime.month, graphEndTime.day);
        int yearRange = timeRange.year;

```

```

int normalizedYearStep = GetNormalizedStep( yearRange );
if (normalizedYearStep==0) normalizedYearStep = 1;

Date datelater(0,1,1);
datelater.year = GetGridStartValue( normalizedYearStep, graphStartDate.year+1 );
if ( datelater < graphStartDate ) {
    GRIDLOG("    ### YEAR STEP_2 :  datelater=" << datelater << "    graphStartDate=" <<
graphStartDate);
    datelater.year += normalizedYearStep;
}

_nbSteps = (unsigned int)tabs((graphEndDate.year - datelater.year) / normalizedYearStep);
if (_nbSteps > _nbMaxSteps) {
    _nbSteps = _nbMaxSteps;
}
else if (_nbSteps==0) _nbSteps = 1;

GRIDLOG("YEAR  range=" << yearRange << "years  normalizedStep = " <<
normalizedYearStep << " years  _nbSteps=" << _nbSteps << "    graphStartDate = " <<
datelater);
for (unsigned int c=0; c<= _nbSteps; ++c) {
    Time tmp = ToTime(datelater);
    _stepDrawingParams[c].stepGraphValue = tmp.Get();
    GRIDLOG("    YEAR STEP :  datelater=" << datelater);
    datelater = ToTime(AddYears(datelater, normalizedYearStep));
}
}
// =====
//    MONTHS
// =====
else if ((timeRange.month >= 7) || (timeRange.year > 0 )) {
    Date graphStartDate( graphStartTime.year, graphStartTime.month, graphStartTime.day);
    Date graphEndDate( graphEndTime.year, graphEndTime.month, graphEndTime.day);
    int monthRange = timeRange.year*12+timeRange.month-1 ;
    Vector<double> monthStep;
    monthStep << 1 << 2 << 3 << 4 << 6 << 12 << 24 << 48;
    int normalizedMonthStep = GetNormalizedStep( monthRange, monthStep );
    if (normalizedMonthStep==0) normalizedMonthStep = 1;

    Date datelater(0,1,1);

    int nbMonths = GetGridStartValue( normalizedMonthStep,
graphStartDate.year*12+graphStartDate.month-1 );
    datelater.year = nbMonths/12;
    datelater.month= nbMonths - datelater.year*12 + 1;
    datelater.day = 1;
    GRIDLOG("    MONTH STEP_1 : " << datelater << "    nbMonths=" << nbMonths << "
graphStartDate=" << graphStartDate);

```

```

if ( datelter < graphStartDate ) {
    GRIDLOG("  #### MONTH STEP_2 :  datelter=" << datelter << "    graphStartDate=" <<
graphStartDate);
    datelter = AddMonths(datelter, normalizedMonthStep);
}

_nbSteps = (unsigned int)tabs(((graphEndDate.year-datelter.year)*12 +
(graphEndDate.month-datelter.month)) / normalizedMonthStep);
if (_nbSteps > _nbMaxSteps) {
    _nbSteps = _nbMaxSteps;
}
else if (_nbSteps==0) _nbSteps = 1;

GRIDLOG("MONTH  range=" << monthRange << " months    normalizedMonthStep = " <<
normalizedMonthStep << " months    _nbSteps=" << _nbSteps << "    graphStartDate = " <<
graphStartDate);
for (unsigned int c=0; c<= _nbSteps; ++c) {
    Time tmp = ToTime(datelter);
    _stepDrawingParams[c].stepGraphValue = tmp.Get();
    GRIDLOG("  MONTH STEP : " << datelter);
    datelter = AddMonths(datelter, normalizedMonthStep);
}
}
// =====
// DAY / HOUR / MINUTES / SECONDS
// =====
else {
    TypeGraphCoord gridStartValue;
    Upp::int64 normalizedStep;
    Vector<double> secondsSteps;
    secondsSteps << 1 << 2 << 5 << 10 << 15 << 20 << 30;
    secondsSteps << MINUTES_sec << MINUTES_sec*2 << MINUTES_sec*3 << MINUTES_sec*5
<< MINUTES_sec*10 << MINUTES_sec*15 << MINUTES_sec*20 << MINUTES_sec*30;
    secondsSteps << HOUR_sec << HOUR_sec*2 << HOUR_sec*3 << HOUR_sec*4 <<
HOUR_sec*6 << HOUR_sec*8 << HOUR_sec*12;
    secondsSteps << DAY_sec << DAY_sec*2 << DAY_sec*3 << DAY_sec*4 << DAY_sec*5 <<
DAY_sec*7 << DAY_sec*14 << DAY_sec*28 << DAY_sec*56;
    normalizedStep = GetNormalizedStep( (timeRange - timeOrigin), secondsSteps );
    if (normalizedStep == 0) normalizedStep = 1;
    gridStartValue = GetGridStartValue( normalizedStep, _coordConverter.getGraphMin() );
    _nbSteps = (unsigned int)tabs((graphEndTime.Get() - gridStartValue) / normalizedStep);

    if (_nbSteps > _nbMaxSteps) {
        _nbSteps = _nbMaxSteps;
    }
    else if (_nbSteps==0) _nbSteps = 1;

    GRIDLOG("D/H/M/S ==>  _nbSteps=" << _nbSteps << "    gridStepValue=" <<

```

```

normalizedStep);
_stepDrawingParams[0].stepGraphValue = gridStartValue;
for (unsigned int c=1; c<_nbSteps+1; ++c)
{
    _stepDrawingParams[c].stepGraphValue = gridStartValue + normalizedStep * c;
    Time time, time2;
    time.Set( _stepDrawingParams[c].stepGraphValue );
    time2.Set( _stepDrawingParams[c-1].stepGraphValue );
    if ((time.month - time2.month) == 1 || (time.month - time2.month)==-11) {
        Time addTime = time;
        addTime.day = 1;
        addTime.minute = 0;
        addTime.second = 0;
        TypeGraphCoord t = addTime.Get();
        if (( t - _stepDrawingParams[c-1].stepGraphValue ) < (normalizedStep/2) )
            _stepDrawingParams[c-1].stepGraphValue = t;
        else
            _stepDrawingParams[c].stepGraphValue = t;
        GRIDLOG("D/H/M/S ==> Modified step at: " << addTime);
    }
}
}
}
}
}
}

```

END_UPP_NAMESPACE