NAG Library Function Document

nag_tsa_cp_binary_user (g13nec)

1 Purpose

nag_tsa_cp_binary_user (g13nec) detects change points in a univariate time series, that is, the time points at which some feature of the data, for example the mean, changes. Change points are detected using binary segmentation for a user-supplied cost function.

2 Specification

```
#include <nag.h>
#include <nagg13.h>

void nag_tsa_cp_binary_user (Integer n, double beta, Integer minss,
     Integer mdepth,
     void (*chgpfn)(Nag_TS_SegSide side, Integer u, Integer w, Integer minss,
          Integer *v, double cost[], Nag_Comm *comm, Integer *info),
     Integer *ntau, Integer tau[], Nag_Comm *comm, NagError *fail)
```

3 Description

Let $y_{1:n} = \{y_j : j = 1, 2, \ldots, n\}$ denote a series of data and $\tau = \{\tau_i : i = 1, 2, \ldots, m\}$ denote a set of m ordered (strictly monotonic increasing) indices known as change points with $1 \le \tau_i \le n$ and $\tau_m = n$. For ease of notation we also define $\tau_0 = 0$. The m change points, τ, split the data into m segments, with the ith segment being of length n_i and containing $y_{\tau_{i-1}+1:\tau_i}$.

Given a cost function, $C(y_{\tau_{i-1}+1:\tau_i})$, nag_tsa_cp_binary_user (g13nec) gives an approximate solution to

$$\underset{m,\tau}{\text{minimize}} \sum_{i=1}^{m} \left(C(y_{\tau_{i-1}+1:\tau_i}) + \beta\right)$$

where β is a penalty term used to control the number of change points. The solution is obtained in an iterative manner as follows:

1. Set $u = 1$, $w = n$ and $k = 0$
2. Set $k = k + 1$. If $k > K$, where K is a user-supplied control parameter, then terminate the process for this segment.
3. Find v that minimizes
$$C(y_{u:v}) + C(y_{v+1:w})$$
4. Test
$$C(y_{u:v}) + C(y_{v+1:w}) + \beta < C(y_{u:w}) \quad (1)$$
5. If inequality (1) is false then the process is terminated for this segment.
6. If inequality (1) is true, then v is added to the set of change points, and the segment is split into two subsegments, $y_{u:v}$ and $y_{v+1:w}$. The whole process is repeated from step 2 independently on each subsegment, with the relevant changes to the definition of u and w (i.e., w is set to v when processing the left hand subsegment and u is set to $v + 1$ when processing the right hand subsegment.

The change points are ordered to give τ.

4 References

Chen J and Gupta A K (2010) *Parameteric Statisical Change Point Analysis With Applications to Genetics Medicine and Finance* **Second Edition** Birkhäuser

5 Arguments

1: **n** – Integer *Input*

On entry: n, the length of the time series.

Constraint: **n** ≥ 2.

2: **beta** – double *Input*

On entry: β, the penalty term.

There are a number of standard ways of setting β, including:

SIC or BIC
: $\beta = p \times \log(n)$.

AIC
: $\beta = 2p$.

Hannan-Quinn
: $\beta = 2p \times \log(\log(n))$.

where p is the number of parameters being treated as estimated in each segment. The value of p will depend on the cost function being used.

If no penalty is required then set $\beta = 0$. Generally, the smaller the value of β the larger the number of suggested change points.

3: **minss** – Integer *Input*

On entry: the minimum distance between two change points, that is $\tau_i - \tau_{i-1} \geq$ **minss**.

Constraint: **minss** ≥ 2.

4: **mdepth** – Integer *Input*

On entry: K, the maximum depth for the iterative process, which in turn puts an upper limit on the number of change points with $m \leq 2^K$.

If $K \leq 0$ then no limit is put on the depth of the iterative process and no upper limit is put on the number of change points, other than that inherent in the length of the series and the value of **minss**.

5: **chgpfn** – function, supplied by the user *External Function*

chgpfn must calculate a proposed change point, and the associated costs, within a specified segment.

The specification of **chgpfn** is:

```
void chgpfn (Nag_TS_SegSide side, Integer u, Integer w, Integer minss,
     Integer *v, double cost[], Nag_Comm *comm, Integer *info)
```

1: **side** – Nag_TS_SegSide *Input*

On entry: flag indicating what **chgpfn** must calculate and at which point of the Binary Segmentation it has been called.

side = Nag_FirstSegCall
: only $C(y_{u:w})$ need be calculated and returned in **cost**[0], neither **v** nor the other elements of **cost** need be set. In this case, $u = 1$ and $w = $ n.

side = Nag_SecondSegCall
all elements of **cost** and **v** must be set. In this case, $u = 1$ and $w = \text{n}$.

side = Nag_LeftSubSeg
the segment, $y_{u:w}$, is a left hand side subsegment from a previous iteration of the Binary Segmentation algorithm. All elements of **cost** and **v** must be set.

side = Nag_RightSubSeg
the segment, $y_{u:w}$, is a right hand side subsegment from a previous iteration of the Binary Segmentation algorithm. All elements of **cost** and **v** must be set.

The distinction between **side** = Nag_LeftSubSeg and Nag_RightSubSeg may allow for **chgpfn** to be implemented in a more efficient manner. See section Section 10 for one such example.

The first call to **chgpfn** will always have **side** = Nag_FirstSegCall and the second call will always have **side** = Nag_SecondSegCall. All subsequent calls will be made with **side** = Nag_LeftSubSeg or Nag_RightSubSeg.

2: **u** – Integer *Input*

On entry: u, the start of the segment of interest.

3: **w** – Integer *Input*

On entry: w, the end of the segment of interest.

4: **minss** – Integer *Input*

On entry: the minimum distance between two change points, as passed to nag_tsa_cp_binary_user (g13nec).

5: **v** – Integer * *Output*

On exit: if **side** = Nag_FirstSegCall then **v** need not be set.

if **side** $\neq$ Nag_FirstSegCall then v, the proposed change point. That is, the value which minimizes

$$\underset{v}{\text{minimize}} C(y_{u:v}) + C(y_{v+1:w})$$

for $v = u + \textbf{minss} - 1$ to $w - \textbf{minss}$.

6: **cost**[**3**] – double *Output*

On exit: costs associated with the proposed change point, v.

If **side** = Nag_FirstSegCall then $\textbf{cost}[0] = C(y_{u:w})$ and the remaining two elements of **cost** need not be set.

If **side** $\neq$ Nag_FirstSegCall then

$\textbf{cost}[0] = C(y_{u:v}) + C(y_{v+1:w})$.

$\textbf{cost}[1] = C(y_{u:v})$.

$\textbf{cost}[2] = C(y_{v+1:w})$.

7: **comm** – Nag_Comm *

Pointer to structure of type Nag_Comm; the following members are relevant to **chgpfn**.

user – double *
iuser – Integer *
p – Pointer

The type Pointer will be `void *`. Before calling nag_tsa_cp_binary_user (g13nec) you may allocate memory and initialize these pointers with various quantities for use by **chgpfn** when called from nag_tsa_cp_binary_user (g13nec) (see Section 3.2.1.1 in the Essential Introduction).

8: **info** – Integer * *Input/Output*

On entry: **info** $= 0$.

On exit: in most circumstances **info** should remain unchanged.

If **info** is set to a strictly positive value then nag_tsa_cp_binary_user (g13nec) terminates with **fail**.**code** = NE_USER_STOP.

If **info** is set to a strictly negative value the current segment is skipped (i.e., no change points are considered in this segment) and nag_tsa_cp_binary_user (g13nec) continues as normal. If **info** was set to a strictly negative value at any point and no other errors occur then nag_tsa_cp_binary_user (g13nec) will terminate with **fail**.**code** = NW_POTENTIAL_PROBLEM.

6: **ntau** – Integer * *Output*

On exit: m, the number of change points detected.

7: **tau**$[dim]$ – Integer *Output*

Note: the dimension, *dim*, of the array **tau** must be at least

$\min\left(\text{ceiling}\frac{\mathbf{n}}{\mathbf{minss}}, 2^{\mathbf{mdepth}}\right)$ when **mdepth** > 0;
$\text{ceiling}\frac{\mathbf{n}}{\mathbf{minss}}$ otherwise.

On exit: the first m elements of **tau** hold the location of the change points. The ith segment is defined by $y_{(\tau_{i-1}+1)}$ to y_{τ_i}, where $\tau_0 = 0$ and $\tau_i = \mathbf{tau}[i-1], 1 \leq i \leq m$.

The remainder of **tau** is used as workspace.

8: **comm** – Nag_Comm *

The NAG communication argument (see Section 3.2.1.1 in the Essential Introduction).

9: **fail** – NagError * *Input/Output*

The NAG error argument (see Section 3.6 in the Essential Introduction).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 3.2.1.2 in the Essential Introduction for further information.

NE_BAD_PARAM

On entry, argument $\langle value\rangle$ had an illegal value.

NE_INT

On entry, **minss** $= \langle value\rangle$.
Constraint: **minss** ≥ 2.

On entry, **n** = ⟨*value*⟩.
Constraint: **n** ≥ 2.

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 3.6.6 in the Essential Introduction for further information.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in the Essential Introduction for further information.

NE_USER_STOP

User requested termination by setting **info** = ⟨*value*⟩.

NW_POTENTIAL_PROBLEM

User requested a segment to be skipped by setting **info** = ⟨*value*⟩.

7 Accuracy

Not applicable.

8 Parallelism and Performance

nag_tsa_cp_binary_user (g13nec) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

Please consult the X06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this function. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

nag_tsa_cp_binary (g13ndc) performs the same calculations for a cost function selected from a provided set of cost functions. If the required cost function belongs to this provided set then nag_tsa_cp_binary (g13ndc) can be used without the need to provide a cost function routine.

10 Example

This example identifies changes in the scale parameter, under the assumption that the data has a gamma distribution, for a simulated dataset with 100 observations. A penalty, β of 3.6 is used and the minimum segment size is set to 3. The shape parameter is fixed at 2.1 across the whole input series.

The cost function used is

$$C(y_{\tau_{i-1}+1:\tau_i}) = 2an_i(\log S_i - \log{(an_i)})$$

where a is a shape parameter that is fixed for all segments and $n_i = \tau_i - \tau_{i-1} + 1$.

10.1 Program Text

```
/* nag_tsa_cp_binary_user (g13nec) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 25, 2014.
 */
/* Pre-processor includes */
#include <stdio.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagg13.h>
#include <nagx02.h>
#include <math.h>

/* Structure to hold extra information that the cost function requires */
typedef struct {
  Integer isinf;
  double shape;
  double *y;
} CostInfo;

/* Functions that are dependent on the cost function used */
void chgpfn(Nag_TS_SegSide side, Integer u, Integer w, Integer minss,
            Integer *v, double cost[], Nag_Comm *comm,
            Integer *info);
double gamma_costfn(double si,Integer n,double shape,Integer *isinf);
Integer get_data(Integer n, Nag_Comm *comm);
void clean_data(Nag_Comm *comm);

int main(void)
{
  /* Integer scalar and array declarations */
  Integer i, minss, n, ntau, mdepth;
  Integer exit_status = 0;
  Integer *tau = 0;

  /* NAG structures and types */
  NagError fail;
  Nag_Comm comm;

  /* Double scalar and array declarations */
  double beta;

  /* Initialise the error structure */
  INIT_FAIL(fail);

  printf("nag_tsa_cp_binary_user (g13nec) Example Program Results\n\n");

  /* Skip heading in data file */
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Read in the problem size, penalty, minimum segment size */
  /* and maximum depth */
#ifdef _WIN32
   scanf_s("%"NAG_IFMT"%lf%"NAG_IFMT"%"NAG_IFMT"%*[^\n]  ",&n,&beta,&minss,&m-
depth);
#else
  scanf("%"NAG_IFMT"%lf%"NAG_IFMT"%"NAG_IFMT"%*[^\n] ",&n,&beta,&minss,&mdepth);
#endif

  /* Read in other data, that (may be) dependent on the cost function */
  get_data(n,&comm);

  /* Allocate output arrays */
  if (!(tau = NAG_ALLOC(n, Integer)))
```

```
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  /* Call nag_tsa_cp_binary_user (g13nec) to detect change points */
  nag_tsa_cp_binary_user(n,beta,minss,mdepth,chgpfn,&ntau,tau,&comm,&fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_tsa_cp_binary_user (g13nec).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

    /* Display the results */
  printf("  -- Change Points --\n");
  printf("  Number     Position\n");
  printf(" ====================\n");
  for (i = 0; i < ntau; i++)
    {
      printf(" %4"NAG_IFMT"       %6"NAG_IFMT"\n",i+1,tau[i]);
    }

 END:
  NAG_FREE(tau);
  clean_data(&comm);

  return(exit_status);
}

void chgpfn(Nag_TS_SegSide side, Integer u, Integer w, Integer minss,
            Integer *v, double cost[], Nag_Comm *comm,
            Integer *info) {
  double shape, ys, this_cost, tcost[2];
  Integer i, nseg1, nseg2, isinf = 0;
  CostInfo *ci;

  ci = (CostInfo *) comm->p;

  /* Get the shape parameter for the gamma distribution from comm */
  shape = ci->shape;

  /* In order to calculate the cost of having a change point at I, we need */
  /* to calculate sum y[j],j=u-1,...,i-1 and sum y[j],j=i,...w-1. We could */
  /* calculate the required values at each call to CHGPFN, but we reuse some */
  /* of these values, so will store the intermediate sums and only */
  /* recalculate the ones we required */

  /* If side = Nag_LeftSubSeg (i.e. we are working with a left hand */
  /* sub-segment), we already have sum y[j-1], j=u,..,i for this value of U, */
  /* so only need the backwards cumulative sum */
  if (side == Nag_FirstSegCall || side == Nag_LeftSubSeg)
    {
      /* ci->user[2*i] = sum y[j-1],j=i+1,...,w */
      ys = 0.0;
      for (i = w; i > u; i--)
        {
          ys += ci->y[i-1];
          comm->user[2*i-3] = ys;
        }
    }

  /* Similarly, if side = Nag_RightSubSeg (i.e. we are working with a right */
  /* hand sub-segment), we already have SUM(Y(I+1:W)) for this value of W, */
  /* so only need the forwards cumulative sum */
  if (side == Nag_FirstSegCall || side == Nag_RightSubSeg)
    {
      /* comm->user[2*i-2] = sum y[j-1], j=u,...,i */
      ys = 0.0;
      for (i = u; i < w + 1; i++)
```

```
        {
          ys += ci->y[i-1];
          comm->user[2*i-2] = ys;
        }
    }

  /* For SIDE = Nag_FirstSegCall we have calculated both sums, and because */
  /* the call with side = Nag_SecondSegCall directly follows we do not need */
  /* to recalculate anything */
  if (side == Nag_FirstSegCall)
    {
      /* Need to calculate the cost for the full segment */
      cost[0] = gamma_costfn(comm->user[2*w-2],w - u + 1,shape,&isinf);
      /* No need to populate the rest of COST or V */
    }
  else
    {
      /* Need to find a potential change point */
      *v = 0;
      cost[0] = 0.0;

      /* Calculate the widths of the sub-segment for the left most potential */
      /* change point, ensuring it has length at least minss */
      nseg1 = minss;
      nseg2 = w - u - minss + 1;

      /* Loop over all possible change point locations (conditional on the */
      /* length of both segments having length >= minss) */
      for (i = u + minss - 1; i < w - minss + 1; i++)
        {
          tcost[0] = gamma_costfn(comm->user[2*i-2],nseg1,shape,&isinf);
          tcost[1] = gamma_costfn(comm->user[2*i-1],nseg2,shape,&isinf);
          if (isinf != 0)
            {
              /* Total cost for change point is -Inf, so have found */
              /* the minimum */
              *v = i;
              cost[1] = tcost[0];
              cost[2] = tcost[1];
              cost[0] = (cost[1] < cost[2]) ? cost[1] : cost[2];
              break;
            }
          else
            {
              this_cost = tcost[0] + tcost[1];
            }

          if (this_cost < cost[0] || *v == 0)
            {
              /* Update the proposed change point location */
              *v = i;
              cost[1] = tcost[0];
              cost[2] = tcost[1];
              cost[0] = this_cost;
            }

          /* Update the size of the next segments */
          nseg1++;
          nseg2--;
        }
    }

  /* Store the ISINF flag */
  ci->isinf = isinf;

  /* Set info nonzero to terminate execution for any reason */
  *info = 0;
}

double gamma_costfn(double si,Integer n,double shape,Integer *isinf) {
  /* Cost function for the gamma distribution */
```

```
  double tmp;

  if (si <= 0.0)
    {
      /* Cost is -Inf */
      *isinf = 1;
      return -X02ALC;
    }
  else
    {
      tmp = ((double) n)*shape;
      return 2.0*tmp*(log(si) - log(tmp));
    }
}

Integer get_data(Integer n, Nag_Comm *comm) {
  /* Read in data that is specific to the cost function */
  double shape;
  Integer i;
  CostInfo *ci;

  /* Allocate some memory for the additional information structure */
  /* This will be pointed to by comm->p */
  comm->p = 0;
  comm->user = 0;
  if (!(ci = NAG_ALLOC(1,CostInfo)))
    {
      printf("Allocation failure\n");
      return -1;
    }

  /* Read in the series of interest */
  if (!(ci->y = NAG_ALLOC(n, double)))
    {
      printf("Allocation failure\n");
      return -1;
    }
  for (i = 0; i < n; i++)
#ifdef _WIN32
    scanf_s("%lf",&(ci->y)[i]);
#else
    scanf("%lf",&(ci->y)[i]);
#endif
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Read in the shape parameter for the Gamma distribution */
#ifdef _WIN32
  scanf_s("%lf%*[^\n] ",&shape);
#else
  scanf("%lf%*[^\n] ",&shape);
#endif

  /* Store the shape parameter in CostInfo structure */
  ci->shape = shape;

  /* Set the warning flag to 0 */
  ci->isinf = 0;

  /* Allocate some workspace into comm that we will be using later */
  if (!(comm->user = NAG_ALLOC(2*n, double)))
    {
      printf("Allocation failure\n");
      return -1;
    }

  /* Store pointer to CostInfo structure in Nag_Comm */
  comm->p = (void *) ci;
```

```
  return 0;
}

void clean_data(Nag_Comm *comm) {
  /* Free any memory allocated in get_data */
  CostInfo *ci;

  if (comm->p) {
    ci = (CostInfo *) comm->p;
    NAG_FREE(ci->y);
  }

  NAG_FREE(comm->p);
  NAG_FREE(comm->user);
}
```

10.2 Program Data

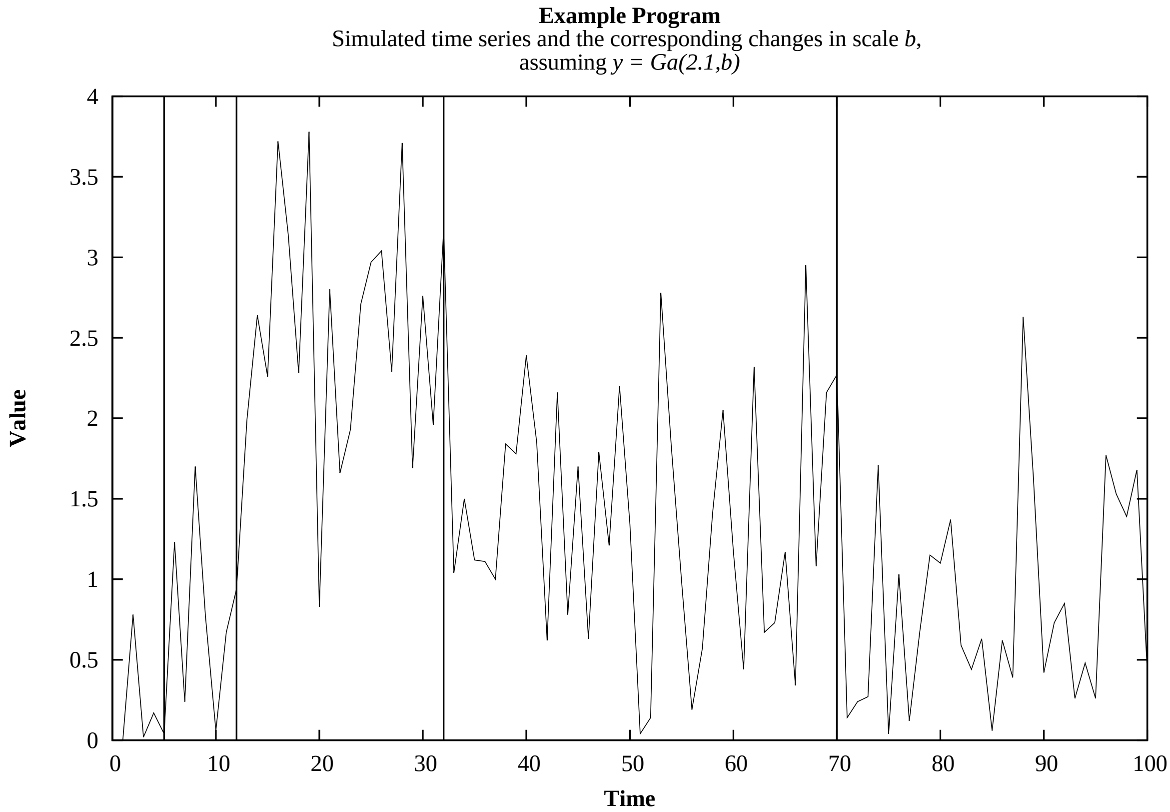
```
nag_tsa_cp_binary_user (g13nec) Example Program Data
100    3.4   3   0 :: n,beta,minss,mdepth
 0.00  0.78  0.02  0.17  0.04  1.23  0.24  1.70  0.77  0.06
 0.67  0.94  1.99  2.64  2.26  3.72  3.14  2.28  3.78  0.83
 2.80  1.66  1.93  2.71  2.97  3.04  2.29  3.71  1.69  2.76
 1.96  3.17  1.04  1.50  1.12  1.11  1.00  1.84  1.78  2.39
 1.85  0.62  2.16  0.78  1.70  0.63  1.79  1.21  2.20  1.34
 0.04  0.14  2.78  1.83  0.98  0.19  0.57  1.41  2.05  1.17
 0.44  2.32  0.67  0.73  1.17  0.34  2.95  1.08  2.16  2.27
 0.14  0.24  0.27  1.71  0.04  1.03  0.12  0.67  1.15  1.10
 1.37  0.59  0.44  0.63  0.06  0.62  0.39  2.63  1.63  0.42
 0.73  0.85  0.26  0.48  0.26  1.77  1.53  1.39  1.68  0.43 :: End of y
2.1                :: shape parameter used in costfn
```

10.3 Program Results

```
nag_tsa_cp_binary_user (g13nec) Example Program Results

  -- Change Points --
  Number     Position
 =====================
    1           5
    2          12
    3          32
    4          70
    5          73
    6         100
```

This example plot shows the original data series and the estimated change points.


Example Program
Simulated time series and the corresponding changes in scale b,
assuming y = Ga(2.1,b)
Value
Time
0
0.5
1
1.5
2
2.5
3
3.5
4
0
10
20
30
40
50
60
70
80
90
100