

# Methods for Smoothing Experimental Data in the smooth Program

Technical Documentation

Version 5.4 | October 13, 2025

## Contents

- [1. Introduction](#)
- [2. Polynomial Fitting \(POLYFIT\)](#)
- [3. Savitzky-Golay Filter \(SAVGOL\)](#)
- [4. Tikhonov Regularization \(TIKHONOV\)](#)
- [5. Method Comparison](#)
- [6. Practical Recommendations](#)
- [7. Usage Examples](#)
- [8. Grid Analysis Module](#)
- [9. Compilation and Installation](#)

## Introduction

The `smooth` program implements three sophisticated methods for smoothing experimental data with the capability of simultaneous derivative computation. Each method has specific properties, advantages, and areas of application.

### General Smoothing Problem

Experimental data often contains random noise:

$$y_{\text{obs}}(x_i) = y_{\text{true}}(x_i) + \varepsilon_i$$

The goal of smoothing is to estimate  $y_{\text{true}}$  while suppressing  $\varepsilon_i$  and preserving physically relevant signal properties.

### Program Structure

```
./smooth [options] data_file
```

#### Basic Parameters:

- `-m {0|1|2}` - method selection (polyfit|savgol|tikhonov)
- `-n N` - smoothing window size (polyfit, savgol)
- `-p P` - polynomial degree (polyfit, savgol, max 12)
- `-l  $\lambda$`  - regularization parameter (tikhonov)
- `-l auto` - automatic  $\lambda$  selection using GCV (tikhonov)
- `-d` - display first derivative in output (optional)
- `-g` - show detailed grid uniformity analysis (optional)

**Note on polynomial degree:** Degrees > 6 may generate numerical stability warnings.

**Note on derivatives:** From version 5.1, first derivative output is optional. Without the `-d` switch, the program outputs only smoothed values. With the `-d` switch, it outputs both smoothed values and first derivatives.

**Note on grid analysis:** The `-g` flag (added in version 5.2) provides detailed grid uniformity statistics helpful for understanding your data and choosing appropriate smoothing parameters.

---

# What's New in Version 5.4

## Major Improvements

#### Tikhonov Regularization (Hybrid Implementation):

- **Automatic discretization selection:** Method automatically chooses between average coefficient (ratio < 2.5) and local spacing (ratio  $\geq 2.5$ ) based on grid uniformity
- **Harmonic mean for better accuracy:** Uses harmonic mean for interval averaging in nearly-uniform grids (more accurate than arithmetic mean)
- **Fixed boundary conditions:** Corrected missing superdiagonal element in local spacing method
- **Improved GCV optimization:** Enhanced with over-fitting penalty and L-curve sanity check for large datasets
- **Better functional computation:** Mathematically correct  $D^2$  discretization matching the matrix formulation

#### Grid Analysis:

- **Detailed reporting with `-g` flag:** Comprehensive grid uniformity statistics including CV, ratio, spacing details
- **Better recommendations:** Program suggests optimal methods based on detected grid characteristics

## API Structure

All smoothing methods have consistent APIs:

```
// Polyfit
PolyfitResult* polyfit_smooth(double *x, double *y, int n,
                             int window_size, int poly_degree);

void free_polyfit_result(PolyfitResult *result);

// Savitzky-Golay
SavgolResult* savgol_smooth(double *x, double *y, int n,
                             int window_size, int poly_degree);

void free_savgol_result(SavgolResult *result);

// Tikhonov - Hybrid implementation in v5.4
TikhonovResult* tikhonov_smooth(double *x, double *y, int n, double lambda);
void free_tikhonov_result(TikhonovResult *result);
```

# Polynomial Fitting (POLYFIT)

## Mathematical Foundations

The POLYFIT method uses local polynomial fitting with least squares method in a sliding window.

**Problem:** For each point  $x_i$ , we fit a polynomial of degree  $p$  to the surrounding  $n$  points:

$$P(x) = a_0 + a_1(x-x_i) + a_2(x-x_i)^2 + \dots + a_p(x-x_i)^p$$

**Optimization criterion:**

$$\min \sum [y_j - P(x_j)]^2 \quad \text{for } j \in [i-n/2, i+n/2]$$

## Construction of Normal Equations

For polynomial coefficients, we solve a system of linear equations:

$$\begin{bmatrix} \sum (x-x_i)^0 & \sum (x-x_i)^1 & \dots & \sum (x-x_i)^p \\ \sum (x-x_i)^1 & \sum (x-x_i)^2 & \dots & \sum (x-x_i)^{p+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum (x-x_i)^p & \sum (x-x_i)^{p+1} & \dots & \sum (x-x_i)^{2p} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_p \end{bmatrix} = \begin{bmatrix} \sum y(x-x_i)^0 \\ \sum y(x-x_i)^1 \\ \vdots \\ \sum y(x-x_i)^p \end{bmatrix}$$

where summation is over points in the window around  $x_i$ .

# Derivative Computation

Derivatives are computed analytically from polynomial coefficients:

```
f(x_i) = a_0  
f'(x_i) = a_1  
f''(x_i) = 2a_2
```

## Edge Handling

At edges, asymmetric windows are used with extrapolation of the fitted polynomial:

```
// For point x_k < x_{n/2}  
f(x_k) =  $\sum_{m=0}^p a_m * (x_k - x_{n/2})^m$ 
```

## Efficient Implementation

The program uses LAPACK routine `dposv` for solving symmetric positive definite systems at each point:

```
// System solution for polynomial coefficients  
dposv_(&uplo, &matrix_size, &nrhs, C, &matrix_size, B, &matrix_size, &info);
```

The normal equations matrix is symmetric and positive definite, making `dposv` optimal for this application.

## Modularized Implementation

```
// polyfit.h  
typedef struct {  
    double *y_smooth;    // Smoothed values  
    double *y_deriv;     // First derivatives  
    int n;               // Number of points  
    int poly_degree;     // Polynomial degree  
    int window_size;     // Window size  
} PolyfitResult;
```

## Characteristics

### Advantages:

- Excellent local approximation
- Analytical computation of derivatives of any order

- Adaptable to changes in curvature
- Good preservation of local extrema
- Works with moderately non-uniform grids

#### Disadvantages:

- Sensitive to outliers
- Boundary effects at edges
- Possible Runge oscillations for high polynomial degrees ( $p > 6$ )
- Numerical instability warnings for degrees  $> 6$

# Savitzky-Golay Filter (SAVGOL)

## Theoretical Foundations

The Savitzky-Golay filter is an optimal linear filter for smoothing and derivatives based on local polynomial regression.

The key innovation is pre-computation of convolution coefficients.

**Fundamental principle:** For given parameters (window size, polynomial degree, derivative order), there exist universal coefficients  $c_k$  such that:

$$f^{(d)}(x_i) = \sum_{k=-n_L}^{n_R} c_k \cdot y_{i+k}$$

## Key Difference from POLYFIT Method

While both SAVGOL and POLYFIT use polynomial approximation, they differ fundamentally in their computational approach:

#### POLYFIT approach:

- For each data point, fits a new polynomial to the surrounding window
- Solves the least squares problem individually for each point
- Coefficients of the polynomial change with each window position
- Computationally intensive:  $O(n \cdot p^3)$

#### SAVGOL approach (Method of Undetermined Coefficients):

- Recognizes that for equidistant grids, the filter coefficients are translation-invariant
- Uses the **method of undetermined coefficients** to pre-compute universal weights
- These weights depend only on the window geometry, not on the actual data values
- Applies the same weights as a linear convolution across all data points
- Computationally efficient:  $O(p^3)$  once, then  $O(n \cdot w)$  for application

## CRITICAL: Grid Uniformity Requirement

**Version 5.3+ Important Feature:** The Savitzky-Golay method now enforces grid uniformity checking.

The mathematical foundation of SG filter assumes **uniformly spaced data points**. The method is based on fitting polynomials in normalized coordinate space where points are at integer positions: {..., -2, -1, 0, 1, 2, ...}.

#### Uniformity Check:

```
CV = std_dev(spacing) / avg(spacing)
```

```
If CV > 0.05: REJECT - Grid too non-uniform for SG
```

```
If CV > 0.01: WARNING - Nearly uniform, proceed with caution
```

```
If CV ≤ 0.01: OK - Grid sufficiently uniform
```

#### What happens when grid is rejected:

```
=====
ERROR: Savitzky-Golay method not suitable for non-uniform grid!
=====

Grid analysis:
  Coefficient of variation (CV) = 0.2341
  Threshold for uniformity = 0.0500

RECOMMENDED ALTERNATIVES:
  1. Use Tikhonov method: -m 2 -l auto
    (Works correctly with non-uniform grids)
  2. Use Polyfit method: -m 0 -n 5 -p 2
    (Local fitting, less sensitive to spacing)
  3. Resample your data to uniform grid before smoothing
```

## The Method of Undetermined Coefficients

The Savitzky-Golay method seeks a linear combination of data points:

$$\hat{y}_0 = c_{-n_l} \cdot y_{-n_l} + \dots + c_0 \cdot y_0 + \dots + c_{n_r} \cdot y_{n_r}$$

where the coefficients  $c_k$  are "undetermined" and must satisfy the condition that the filter exactly reproduces polynomials up to degree  $p$ .

**The key insight:** For a given window configuration and polynomial degree, these coefficients can be determined once and applied universally - but only on uniform grids!

## Coefficient Derivation

Coefficients are derived from the condition that the filter must exactly reproduce polynomials up to degree  $p$ .

#### Moment conditions:

$$\sum_{j=-n_L}^{n_R} c_j \cdot j^m = \delta_{\{m,d\}} \cdot d! \quad \text{for } m = 0, 1, \dots, p$$

where:

- $\delta_{\{m,d\}}$  is the Kronecker delta
- $d$  is the derivative order
- $d!$  is factorial

This leads to a system of linear equations where the unknowns are the filter coefficients  $c_j$ .

## Matrix Formulation

We solve a system of linear equations:

$$\begin{bmatrix} 1 & -n_L & (-n_L)^2 & \dots & (-n_L)^p \\ 1 & -n_L+1 & (-n_L+1)^2 & \dots & (-n_L+1)^p \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & n_R & n_R^2 & \dots & n_R^p \end{bmatrix} \begin{bmatrix} c_{-n_L} \\ c_{-n_L+1} \\ \vdots \\ c_{n_R} \end{bmatrix} = \begin{bmatrix} \delta_{\{0,d\}} \cdot 0! \\ \delta_{\{1,d\}} \cdot 1! \\ \vdots \\ \delta_{\{p,d\}} \cdot p! \end{bmatrix}$$

## Computational Efficiency

The brilliance of the Savitzky-Golay approach becomes apparent when processing large datasets:

**Example for 10,000 data points, window size 21, polynomial degree 4:**

- **POLYFIT:** Must solve 10,000 separate 5×5 linear systems
- **SAVGOL:** Solves only ONE 5×5 system, then performs 10,000 simple weighted sums

This difference explains why SAVGOL is preferred for real-time signal processing and large datasets, while maintaining the same mathematical accuracy as POLYFIT **for uniform grids**.

## Efficient Implementation

The program uses LAPACK routine `dposv` for solving the symmetric positive definite system when computing filter coefficients:

```
// Solve linear system for Savitzky-Golay coefficients
dposv_(&uplo, &matrix_size, &nrhs, A, &matrix_size, B, &matrix_size, &info);
```

# Modularized Implementation

```
// savgol.h
typedef struct {
    double *y_smooth;    // Smoothed values
    double *y_deriv;     // First derivatives
    int n;               // Number of points
    int poly_degree;     // Polynomial degree
    int window_size;     // Window size
} SavgolResult;

// Coefficient computation
void savgol_coefficients(int nl, int nr, int poly_degree,
                        int deriv_order, double *c);
```

## Optimal Properties

The Savitzky-Golay filter minimizes approximation error in the least squares sense and maximizes signal-to-noise ratio for polynomial signals **on uniform grids**.

## Characteristics

### Advantages:

- Optimal for polynomial signals on uniform grids
- Excellent preservation of moments and peak areas
- Efficient implementation (convolution)
- Minimal phase distortion
- Simultaneous computation of functions and derivatives

### Disadvantages:

- **Requires uniform grid** - automatically rejected if CV > 0.05
- Fixed coefficients for entire window
- May introduce oscillations at sharp edges
- Limited adaptability
- Numerical warnings for degrees > 6

---

# Tikhonov Regularization (TIKHONOV)

## Theoretical Foundation



Tikhonov regularization solves the ill-posed inverse smoothing problem using a variational approach. We seek a function minimizing the functional:

#### Continuous formulation:

$$J[u] = \underbrace{\int (y(x) - u(x))^2 dx}_{\text{Data fidelity term}} + \underbrace{\lambda \int (u'(x))^2 dx}_{\text{Smoothness penalty}}$$

#### Discrete formulation:

$$J[u] = ||y - u||^2 + \lambda ||D^2 u||^2$$

where:

- $||y - u||^2 = \sum (y_i - u_i)^2$  is the **data fidelity term**
- $||D^2 u||^2 = \sum (D^2 u_i)^2$  is the **regularization term** (smoothness penalty)
- $\lambda$  is the **regularization parameter** controlling the balance
- $D^2$  is the discrete second derivative operator

## The Regularization Parameter $\lambda$

The parameter  $\lambda$  is the **heart of Tikhonov regularization** - it controls the balance between fitting the data and smoothing the result.

### Physical Interpretation

$\lambda = 0$ : No smoothing,  $u = y$  (exact data fit)  
 $J[u] = ||y - u||^2$  only

$\lambda \rightarrow \infty$ : Maximum smoothing,  $u \rightarrow$  straight line  
 $J[u] \approx \lambda ||D^2 u||^2$  dominates

$\lambda$  optimal: Balanced between data fit and smoothness  
Both terms contribute meaningfully

### Mathematical Role

The minimization of  $J[u]$  leads to:

$$(I + \lambda D^T D)u = y$$

#### Effect of $\lambda$ on the solution:

- **Small  $\lambda$  (< 0.01):** Matrix  $\approx I \rightarrow$  solution  $u \approx y$  (minimal smoothing)
- **Large  $\lambda$  (> 1.0):** Matrix  $\approx \lambda D^4$   $\rightarrow$  strong curvature penalty (heavy smoothing)
- **Optimal  $\lambda$ :** Matrix components balanced  $\rightarrow$  noise removed, signal preserved

## Frequency Domain Interpretation

In Fourier space, Tikhonov acts as a low-pass filter:

$$\hat{H}(\omega) = 1 / (1 + \lambda \omega^4)$$

where  $\omega$  is spatial frequency.

### Effect:

- **Low frequencies (slow variations):**  $\hat{H} \approx 1 \rightarrow$  preserved
- **High frequencies (noise, rapid variations):**  $\hat{H} \approx 1/(\lambda \omega^4) \rightarrow$  attenuated
- **Cutoff frequency:**  $\omega_c \propto \lambda^{-1/4}$

### This means:

Larger  $\lambda \rightarrow$  Lower cutoff  $\rightarrow$  More aggressive low-pass filtering  $\rightarrow$  Smoother result  
Smaller  $\lambda \rightarrow$  Higher cutoff  $\rightarrow$  Less filtering  $\rightarrow$  Result closer to data

## Practical Guidelines for $\lambda$ Selection

### 1. Automatic Selection (RECOMMENDED):

```
./smooth -m 2 -l auto data.txt
```

Uses Generalized Cross Validation (GCV) to find optimal  $\lambda$ .

### 2. Manual Selection:

Data Characteristics	Recommended $\lambda$	Reasoning
Low noise, important details	0.001 - 0.01	Preserve features
Moderate noise	0.01 - 0.1	Balanced (default: 0.1)
High noise	0.1 - 1.0	Strong smoothing
Very noisy, global trends	1.0 - 10.0	Maximum smoothing

### 3. Iterative Refinement:

```
# Start with automatic
./smooth -m 2 -l auto data.txt
```

```
# If result is over-smoothed (details lost):
./smooth -m 2 -l 0.01 data.txt
```

```
# If result is under-smoothed (still noisy):
./smooth -m 2 -l 1.0 data.txt
```

#### 4. Diagnostic Criteria:

The program outputs functional components:

```
# Functional J = 1.234e+02 (Data: 5.67e+01 + Regularization: 6.67e+01)
# Data/Total ratio = 0.460, Regularization/Total ratio = 0.540
```

#### Good balance indicators:

- Data term: 30-70% of total functional
- Regularization term: 30-70% of total functional

#### Warning signs:

- Data term > 95%: Under-smoothed ( $\lambda$  too small)
- Regularization term > 95%: Over-smoothed ( $\lambda$  too large)

#### 5. Grid-Dependent Considerations:

For non-uniform grids with ratio  $h_{\max}/h_{\min} > 5$ :

```
# Start with more conservative (larger)  $\lambda$ 
./smooth -m 2 -l 0.5 nonuniform_data.txt

# GCV may be less accurate - check results visually
./smooth -m 2 -l auto nonuniform_data.txt
```

### $\lambda$ and Grid Spacing

The effective regularization strength depends on grid spacing:

Effective strength  $\propto \lambda / h^2_{\text{avg}}$

Same  $\lambda$  on finer grid  $\rightarrow$  weaker smoothing

Same  $\lambda$  on coarser grid  $\rightarrow$  stronger smoothing

For dimensional consistency,  $\lambda$  has units [Length<sup>2</sup>].

## Second Derivative Discretization (Hybrid Method v5.4)

**Version 5.4 Implementation:** Automatic selection between two discretization schemes based on grid uniformity.

### Grid Uniformity Detection

$\text{ratio} = h_{\text{max}} / h_{\text{min}}$

$\text{ratio} < 2.5$ : Nearly uniform  $\rightarrow$  Average Coefficient Method

$\text{ratio} \geq 2.5$ : Highly non-uniform  $\rightarrow$  Local Spacing Method

### Method 1: Average Coefficient (for ratio < 2.5)

Used for uniform and mildly non-uniform grids. More robust numerically.

**Discretization:** For interior point  $i$  with neighbors at spacing  $h_{\text{left}}$  and  $h_{\text{right}}$ , use **harmonic mean**:

$h_{\text{harm}} = 2 \cdot h_{\text{left}} \cdot h_{\text{right}} / (h_{\text{left}} + h_{\text{right}})$

$D^2 u_i \approx (u_{i-1} - 2u_i + u_{i+1}) / h_{\text{harm}}^2$

#### Why harmonic mean?

- More accurate than arithmetic mean for averaging intervals
- Gives greater weight to smaller spacing (physically correct)
- For  $h_{\text{left}} = h_{\text{right}}$ , reduces to standard formula

#### Matrix construction:

$c = \lambda \cdot \Sigma(1/h_i^2) / (n-1)$  (average coefficient)

$A[i,i] = 1 + 2c$  (interior points)

$A[i,i\pm1] = -c$  (off-diagonals)

### Method 2: Local Spacing (for ratio $\geq 2.5$ )

Used for highly non-uniform grids. More accurate for variable spacing.

**Discretization:** For point  $i$  with left spacing  $h_1 = x[i] - x[i-1]$  and right spacing  $h_2 = x[i+1] - x[i]$ :

$$D^2 u_i \approx (2/(h_1+h_2)) \cdot [u_{i-1}/h_1 - u_i \cdot (1/h_1 + 1/h_2) + u_{i+1}/h_2]$$

This is the **correct second derivative formula** for non-uniform grids derived from Taylor expansion.

**Matrix construction:**

$$w = 2\lambda / (h_1 + h_2)$$

$$A[i,i] = 1 + w \cdot (1/h_1 + 1/h_2)$$

$$A[i,i-1] = -w/h_1$$

$$A[i,i+1] = -w/h_2$$

**The resulting matrix is:**

- Symmetric
- Positive definite
- Tridiagonal (bandwidth = 1)

## Boundary Conditions

Natural boundary conditions (second derivative = 0 at ends):

**Left boundary (i=0):**

$$D^2 u_0 \approx (u_1 - u_0) / h_0^2$$

$$A[0,0] += \lambda/h_0^2$$

$$A[0,1] += -\lambda/h_0^2$$

**Right boundary (i=n-1):**

$$D^2 u_{n-1} \approx (u_{n-1} - u_{n-2}) / h_{n-1}^2$$

$$A[n-1,n-1] += \lambda/h_{n-1}^2$$

**Critical fix in v5.4:** The boundary superdiagonal element  $A[0,1]$  was missing in previous versions, causing isolation of the first point. This is now corrected.

## Functional Computation

The actual value of the minimized functional is computed for diagnostic purposes:

**Data term:**

$$||y - u||^2 = \sum (y_i - u_i)^2$$

**Regularization term (must match matrix formulation!):**

**For average coefficient method:**

$$||D^2u||^2 = \sum_{\text{interior}} [(u_{i-1} - 2u_i + u_{i+1})/h_{\text{harm}}]^2 \\ + 0.5 \cdot [(u_1 - u_0)/h_0]^2 \\ + 0.5 \cdot [(u_{n-1} - u_{n-2})/h_{n-1}]^2$$

**For local spacing method:**

$$||D^2u||^2 = \sum_{\text{interior}} [D^2u_i]^2 \cdot (h_1 + h_2)/2 \\ + 0.5 \cdot [(u_1 - u_0)/h_0]^2 \cdot h_0 \\ + 0.5 \cdot [(u_{n-1} - u_{n-2})/h_{n-1}]^2 \cdot h_{n-1}$$

Note the **weighting factors** in local spacing method for proper integration over non-uniform grid.

**Total functional:**

$$J[u] = ||y - u||^2 + \lambda ||D^2u||^2$$

## Variational Approach

The minimum of functional  $J[u]$  satisfies the Euler-Lagrange equation:

$$\partial J / \partial u_i = 0 \implies -2(y_i - u_i) + 2\lambda (D^T D u)_i = 0$$

which leads to the linear system:

$$(I + \lambda D^T D)u = y$$

## Matrix Representation

Matrix  $A = I + \lambda D^T D$  is:

- Symmetric
- Positive definite

- Tridiagonal (banded with bandwidth 1)

This structure allows efficient solution using LAPACK's banded solver `dpbsv`.

## Generalized Cross Validation (GCV)

For automatic  $\lambda$  selection (`-1 auto`), we minimize the GCV criterion:

$$\text{GCV}(\lambda) = n \cdot \text{RSS}(\lambda) / (n - \text{tr}(H_\lambda))^2$$

where:

- $\text{RSS}(\lambda) = ||y - u_\lambda||^2$  is the residual sum of squares
- $H_\lambda = (I + \lambda D^T D)^{-1}$  is the influence matrix (smoother matrix)
- $\text{tr}(H_\lambda)$  is the trace (effective number of parameters)

### Interpretation:

- $\text{tr}(H_\lambda)$  measures model complexity (degrees of freedom)
- Small  $\lambda$ :  $\text{tr}(H) \approx n$  (interpolation, overfitting)
- Large  $\lambda$ :  $\text{tr}(H) \approx 2$  (straight line, underfitting)
- Optimal  $\lambda$ : minimizes prediction error

### Trace estimation using eigenvalues:

For uniform grids with natural boundary conditions:

$$\text{tr}(H_\lambda) \approx \sum_{k=1}^n 1 / (1 + \lambda \mu_k)$$

where eigenvalues:

$$\theta_k = \pi k / n$$

$$\mu_k = 4 \cdot \sin^2(\theta_k / 2) / h^2$$

**Note:** This approximation is exact for uniform grids but approximate for non-uniform grids. For highly non-uniform grids (ratio > 5), the program issues a warning.

## Enhanced GCV in v5.4

### Over-fitting penalty:

If  $\text{tr}(H)/n > 0.7$ :

$$\text{GCV\_modified} = \text{GCV} \cdot \exp(10 \cdot (\text{tr}(H)/n - 0.7))$$

This exponential penalty prevents selection of too-small  $\lambda$  that would lead to overfitting.

### L-curve backup (for $n > 20000$ ):

For very large datasets, GCV trace approximation may be inaccurate. The program also computes the L-curve (plot of  $\|D^2u\|^2$  vs  $\|y-u\|^2$ ) and finds the corner point with maximum curvature:

$$\kappa = |x'y'' - y'x''| / (x'^2 + y'^2)^{3/2}$$

where:

$$x = \log(\|y - u\|^2)$$

$$y = \log(\|D^2u\|^2)$$

If GCV and L-curve disagree significantly, the program uses the more conservative (larger)  $\lambda$ .

## Efficient Implementation

The program uses LAPACK routine `dpbsv` for solving symmetric positive definite banded systems:

```
// Banded matrix storage (LAPACK column-major format)
AB[0,j] = superdiagonal elements
AB[1,j] = diagonal elements

// System solution
dpbsv_(&uplo, &n, &kd, &nrhs, AB, &ldab, b, &n, &info);
```

### Complexity:

- Memory:  $O(n)$  for banded storage
- Time:  $O(n)$  for factorization and back-substitution

This is **optimal** for tridiagonal systems.

## Hybrid Implementation (v5.4)



```

typedef struct {
    double *y_smooth;           // Smoothed values
    double *y_deriv;            // First derivatives
    double lambda;               // Used parameter
    int n;                       // Number of points
    double data_term;            //  $\|y - u\|^2$ 
    double regularization_term;  //  $\lambda \|D^2 u\|^2$ 
    double total_functional;     //  $J[u]$ 
} TikhonovResult;

// Main function
TikhonovResult* tikhonov_smooth(double *x, double *y, int n, double lambda);

// Automatic  $\lambda$  selection
double find_optimal_lambda_gcv(double *x, double *y, int n);

// Memory cleanup
void free_tikhonov_result(TikhonovResult *result);

```

## Characteristics

### Advantages:

- Global optimization with theoretical foundation
- Flexible balance between data fidelity and smoothness (controlled by  $\lambda$ )
- Robust to outliers (quadratic penalty less sensitive than least squares)
- Efficient for large datasets ( $O(n)$  memory and time)
- **Automatic  $\lambda$  selection via GCV** - no guessing needed
- **Excellent for non-uniform grids** - correct discretization automatic
- **Unified approach** - same algorithm for uniform and non-uniform grids
- Works well for noisy data with global trends

### Disadvantages:

- Single global parameter  $\lambda$  (cannot vary locally)
- May suppress local details if  $\lambda$  too large
- GCV may fail for some data types (especially highly non-uniform grids)
- Requires LAPACK library
- Boundary effects if data has discontinuities at edges

---

## Method Comparison

## Computational Complexity

Method	Time	Memory	Scalability
POLYFIT	$O(n \cdot p^3)$	$O(p^2)$	Good for small $p$
SAVGOL	$O(p^3) + O(n \cdot w)$	$O(w)$	Excellent for large $n$
TIKHONOV	$O(n)$	$O(n)$	Excellent

*Note:  $w$  = window size,  $p$  = polynomial degree ( $\leq 12$ ),  $n$  = number of data points.*

## Smoothing Quality

Property	POLYFIT	SAVGOL	TIKHONOV
Local adaptability	*****	****	**
Extreme preservation	****	*****	***
Noise robustness	***	****	*****
Derivative quality	*****	*****	***
Boundary behavior	**	***	****
Non-uniform grids	***	X	*****
Ease of use (v5.4)	****	****	*****
Parameter selection	Manual	Manual	Auto (GCV)

**Key:** X = Not suitable (automatically rejected)

## Grid Type Compatibility

Grid Type	POLYFIT	SAVGOL	TIKHONOV
Perfectly uniform ( $CV < 0.01$ )	✓	✓	✓
Nearly uniform ( $CV < 0.05$ )	✓	⚠	✓
Moderately non-uniform ( $0.05 < CV < 0.2$ )	✓	X	✓
Highly non-uniform ( $CV > 0.2$ )	⚠	X	✓
Very large ratio ( $h_{\max}/h_{\min} > 10$ )	⚠	X	✓*

**Legend:**

- ✓ = Recommended
- ⚠ = Usable with caution
- X = Rejected or not recommended
- \* = Uses local spacing method automatically

---

## Practical Recommendations

### Method Selection by Data Type

POLYFIT - when:

- Data has variable curvature
- You need to preserve local details
- You have moderately noisy data
- You want highest quality derivatives
- Grid has moderate spacing variations
- You need local adaptability

### SAVGOL - when:

- **Grid is uniform (CV < 0.05)** - automatically checked!
- You want mathematically optimal linear smoothing for polynomial signals
- Data contains periodic or oscillatory components that need preservation
- You need excellent peak shape preservation (areas, moments)
- You want minimal phase distortion in the smoothed signal
- You're processing time series or spectroscopic data on uniform grids
- You need simultaneous high-quality function and derivative estimation
- Computational efficiency is critical (large datasets)

### TIKHONOV - when:

- **Grid is non-uniform** - works perfectly automatically!
- Data is very noisy
- You need global consistency
- **You want automatic parameter selection ( $\lambda$  auto)** - highly recommended!
- You prefer global optimization approaches over local fitting
- You want robust handling of outliers
- You want the simplest workflow (one parameter, automatic selection)
- You need to process very large datasets efficiently
- **You're not sure which method to use** - Tikhonov with `-l auto` is safest!

## Parameter Selection

### Window size (n) for POLYFIT/SAVGOL:

$n = 2*k + 1$  (odd number)

Recommendations:

- Low noise:  $n = 5-9$
- Medium noise:  $n = 9-15$
- High noise:  $n = 15-25$

Rule of thumb:  $n \approx 2p + 3$

### Polynomial degree (p):

- Linear trends:  $p = 1-2$
- Smooth curves:  $p = 2-3$
- Complex signals:  $p = 3-4$
- Advanced applications:  $p = 5-8$
- Maximum:  $p \leq 12$
- Recommended maximum:  $p < n/2$

Note: Degrees  $> 6$  may cause numerical instability warnings.

## Lambda ( $\lambda$ ) for TIKHONOV:

### **\*\*RECOMMENDED:\*\***

- Auto selection: `-l auto` (uses GCV optimization)

### **\*\*MANUAL SELECTION:\*\***

Starting points by noise level:

- Low noise:  $\lambda = 0.001 - 0.01$
- Medium noise:  $\lambda = 0.01 - 0.1$  (default: 0.1)
- High noise:  $\lambda = 0.1 - 1.0$
- Very noisy data:  $\lambda = 1.0 - 10.0$

Full range:  $10^{-6}$  to  $10^3$

### **\*\*ITERATIVE REFINEMENT:\*\***

1. Start with `-l auto`
2. Check functional balance (should be 30-70% each)
3. If over-smoothed: decrease  $\lambda$  by factor of 10
4. If under-smoothed: increase  $\lambda$  by factor of 10
5. Repeat until satisfied

### **\*\*GRID-DEPENDENT:\*\***

For non-uniform grids (ratio  $> 5$ ):

- Start more conservative (larger  $\lambda$ )
- GCV may be less accurate - check visually

---

# Usage Examples

## Basic Syntax

```

# Polynomial fitting (smoothed values only)
./smooth -m 0 -n 7 -p 2 data.txt

# Polynomial fitting with derivatives
./smooth -m 0 -n 7 -p 2 -d data.txt

# Savitzky-Golay (smoothed values only)
# NOTE: Will be rejected if grid is non-uniform!
./smooth -m 1 -n 9 -p 3 data.txt

# Savitzky-Golay with derivatives
./smooth -m 1 -n 9 -p 3 -d data.txt

# Tikhonov with automatic  $\lambda$  (RECOMMENDED)
./smooth -m 2 -l auto data.txt

# Tikhonov with automatic  $\lambda$  and derivatives
./smooth -m 2 -l auto -d data.txt

# Tikhonov with manual  $\lambda$ 
./smooth -m 2 -l 0.01 data.txt

# Tikhonov with manual  $\lambda$  and derivatives
./smooth -m 2 -l 0.01 -d data.txt

# Grid analysis (works with any method)
./smooth -m 2 -l auto -g data.txt

```

## Output Format

**Without -d flag:**

```

# Data smooth - Tikhonov regularization with lambda = 1e-01
# Functional J = 1.234e+02 (Data: 5.67e+01 + Regularization: 6.67e+01)
# Data/Total ratio = 0.460, Regularization/Total ratio = 0.540
#      x          y
# 0.00000E+00  1.00000E+00
# 1.00000E+00  2.71828E+00
# ...

```

**With -d flag:**

```
# Data smooth - Tikhonov regularization with lambda = 1e-01
# Functional J = 1.234e+02 (Data: 5.67e+01 + Regularization: 6.67e+01)
# Data/Total ratio = 0.460, Regularization/Total ratio = 0.540
#      x          y          y'
  0.00000E+00  1.00000E+00  1.00000E+00
  1.00000E+00  2.71828E+00  2.71828E+00
  ...
```

**With -g flag (grid analysis):**

```
# =====
# GRID UNIFORMITY ANALYSIS
# =====
# Grid uniformity analysis:
#   n = 1000 points
#   h_min = 9.500000e-03, h_max = 1.200000e-02, h_avg = 1.000000e-02
#   h_max/h_min = 1.26, CV = 0.052
#   Grid type: NON-UNIFORM
#   Uniformity score: 0.94
#   Standard deviation: 5.200000e-04
#   Detected clusters: 0
#   Recommendation: Grid is nearly uniform - standard methods work well
# =====
#
# Using Average Coefficient Method (h_max/h_min = 1.26)
# GCV optimization for n=1000 points (h_max/h_min = 1.26)
# ...
```

## Working with Non-uniform Grids (v5.4)

```
# First, analyze your grid
./smooth -m 2 -l auto -g nonuniform_data.txt

# If ratio < 2.5, program uses average coefficient method
# Output: "Using Average Coefficient Method (h_max/h_min = 1.85)"

# If ratio ≥ 2.5, program uses local spacing method
# Output: "Using Local Spacing Method (h_max/h_min = 8.42)"

# Automatic selection - no user intervention needed!
# Just use the same command:
./smooth -m 2 -l auto nonuniform_data.txt

# For very non-uniform grids (ratio > 5), you may see:
# "WARNING: Highly non-uniform grid detected!"
# "GCV trace approximation may be less accurate."
# In this case, try manual  $\lambda$  or check results visually.
```

## Typical Workflow

### 1. Quick data exploration with grid analysis:

```
# Check grid uniformity
./smooth -m 2 -l auto -g data.txt > smooth_data.txt

# Review output comments for:
# - Grid uniformity (CV, ratio)
# - Which discretization method was used
# - Functional balance (data vs regularization)
```

### 2. Choose method based on grid:

```
# For uniform grids (CV < 0.05):
./smooth -m 1 -n 9 -p 3 -d data.txt

# For non-uniform grids:
./smooth -m 2 -l auto -d data.txt
```

### 3. Refine $\lambda$ if needed:

```
# If automatic  $\lambda$  gives over-smoothing:  
./smooth -m 2 -l 0.01 -d data.txt  
  
# If under-smoothing:  
./smooth -m 2 -l 1.0 -d data.txt
```

#### 4. For publication graphics:

```
# Final smoothing with derivatives  
./smooth -m 2 -l auto -d data.txt > publication_data.txt  
  
# Check functional balance in output comments  
# Ideal: both terms contribute 30-70%
```

---

## Grid Analysis Module

The `grid_analysis` module provides comprehensive analysis of input data and helps optimize smoothing parameters.

### Main Functions

```
// Complete grid analysis  
GridAnalysis* analyze_grid(double *x, int n, int store_spacings);  
  
// Quick uniformity check  
int is_uniform_grid(double *x, int n, double *h_avg, double tolerance);  
  
// Method recommendation  
const char* get_grid_recommendation(GridAnalysis *analysis);  
  
// Optimal window size  
int optimal_window_size(GridAnalysis *analysis, int min_window, int max_window);
```

### GridAnalysis Structure



```
typedef struct {
    double h_min;           // Minimum spacing
    double h_max;           // Maximum spacing
    double h_avg;           // Average spacing
    double h_std;           // Standard deviation
    double ratio_max_min;   // h_max/h_min ratio
    double cv;              // Coefficient of variation
    double uniformity_score; // Uniformity score (0-1)
    int is_uniform;          // 1 = uniform, 0 = non-uniform
    int n_clusters;          // Number of detected clusters
    int reliability_warning; // Reliability warning
    char warning_msg[512];   // Warning text
    double *spacings;        // Array of spacings (optional)
    int n_points;            // Number of points
    int n_intervals;         // Number of intervals (n-1)
} GridAnalysis;
```

## Example Analysis Output

```
# =====
# GRID UNIFORMITY ANALYSIS
# =====
# Grid uniformity analysis:
#   n = 1000 points
#   h_min = 1.000000e-02, h_max = 1.000000e-01, h_avg = 5.500000e-02
#   h_max/h_min = 10.00, CV = 0.450
#   Grid type: NON-UNIFORM
#   Uniformity score: 0.35
#   Standard deviation: 2.475000e-02
#   Detected clusters: 2
#   Recommendation: High non-uniformity - adaptive methods recommended
# WARNING: HIGH grid non-uniformity: h_max/h_min = 10.0
# Adaptive methods are strongly recommended.
# Consider using smaller regularization parameters.
#
# WARNING: 2 abrupt spacing changes detected (possible data clustering).
# Standard methods may over-smooth clustered regions.
# =====
```

## Grid Uniformity Thresholds

```
CV < 0.01:   Perfectly uniform - all methods work optimally
             → POLYFIT, SAVGOL, TIKHONOV all excellent

CV < 0.05:   Nearly uniform - SAVGOL works with warning
             → SAVGOL may show warning but works
             → POLYFIT and TIKHONOV work fine

0.05 ≤ CV < 0.20: Moderately non-uniform
             → SAVGOL rejected automatically
             → POLYFIT usable
             → TIKHONOV recommended (uses average coef if ratio < 2.5)

CV ≥ 0.20:   Highly non-uniform
             → SAVGOL rejected
             → POLYFIT with caution
             → TIKHONOV strongly recommended (may use local spacing)

RATIO:
ratio < 2.5:  Tikhonov uses average coefficient method
ratio ≥ 2.5:  Tikhonov uses local spacing method
ratio > 10:   Warning issued, GCV may be less accurate
```

---

# Compilation and Installation

## Requirements

- C compiler (gcc, clang)
- LAPACK and BLAS libraries
- Make (optional)

## Compilation using Make

```
# Standard compilation
make

# Debug build
make debug

# Clean
make clean

# Install to user's home directory
make install-user

# Install to system (requires root)
make install
```

## Manual Compilation

```
# Standard compilation
gcc -o smooth smooth.c polyfit.c savgol.c tikhonov.c \
    grid_analysis.c decomment.c -llapack -lblas -lm -O2

# With warnings
gcc -Wall -Wextra -pedantic -o smooth smooth.c polyfit.c savgol.c \
    tikhonov.c grid_analysis.c decomment.c -llapack -lblas -lm -O2
```

## File Structure

```
smooth/
├─ smooth.c           # Main program (v5.2: added -g flag)
├─ polyfit.c/h        # Polynomial fitting module
├─ savgol.c/h         # Savitzky-Golay module (v5.3: with uniformity check)
├─ tikhonov.c/h       # Tikhonov module (v5.4: hybrid implementation)
├─ grid_analysis.c/h  # Grid analysis
├─ decomment.c/h      # Comment removal
├─ revision.h         # Program version
├─ Makefile           # Build system
└─ README.md          # This documentation
```

---

## Conclusion

The `smooth` program v5.4 provides three complementary smoothing methods in a modular architecture with advanced input data analysis:

- **POLYFIT** - local polynomial approximation using least squares method
- **SAVGOL** - optimal linear filter with pre-computed coefficients (uniform grids only)
- **TIKHONOV** - global variational method with hybrid automatic discretization
- **GRID\_ANALYSIS** - automatic analysis and method recommendation

## Version 5.4 Highlights

### Major Improvements:

1. **Tikhonov hybrid implementation** - automatic selection between average coefficient (ratio  $< 2.5$ ) and local spacing (ratio  $\geq 2.5$ ) methods
2. **Harmonic mean** - more accurate interval averaging for nearly-uniform grids
3. **Fixed boundary conditions** - corrected missing superdiagonal element
4. **Enhanced GCV** - over-fitting penalty and L-curve backup for large datasets
5. **Better diagnostics** - functional balance reporting,  $\lambda$  selection guidance
6. **Grid analysis with `-g`** - detailed uniformity statistics for parameter optimization

## When to Use Each Method

### Quick Decision Tree:

```
Is your grid uniform (CV < 0.05)?
├─ YES: Use SAVGOL for best computational efficiency
|      smooth -m 1 -n 9 -p 3 -d data.txt
|
└─ NO:  Use TIKHONOV for correct handling
       smooth -m 2 -l auto -d data.txt

Need local adaptability?
└─ Use POLYFIT regardless of grid
   smooth -m 0 -n 7 -p 2 -d data.txt

Not sure?
└─ Use TIKHONOV with automatic  $\lambda$  - safest choice!
   smooth -m 2 -l auto -g -d data.txt
```

Each method has a strong mathematical foundation and is optimized for specific data types. The program provides automatic guidance on method selection and parameters, with extensive diagnostics to ensure correct usage.

## Best Practices

1. **Always check grid first:** Use `-g` flag to understand your data
  2. **Start with automatic:** Use `-l auto` for Tikhonov, let GCV find optimal  $\lambda$
  3. **Check functional balance:** Look for 30-70% split between data and regularization terms
  4. **Iterate if needed:** Adjust  $\lambda$  manually if automatic selection doesn't satisfy requirements
  5. **Use derivatives wisely:** Add `-d` only when needed - cleaner output without it
  6. **Understand the trade-off:** More smoothing (larger  $\lambda$ ) = more noise reduction but less detail
- 

**Document revision:** 2025-10-13

**Program version:** smooth v5.4

**Dependencies:** LAPACK, BLAS

**License:** See source files