PHCpack Documentation

Release 2.4.47

Jan Verschelde

Oct 17, 2017

Contents

CHAPTER 1

Getting Started

This documentation describes the software package PHCpack.

1.1 What is PHCpack?

PHCpack implements a collection of algorithms to solve polynomial systems by homotopy continuation methods.

On input is a sequence of polynomials in several variables, on output are the solutions to the polynomial system given on input. The computational complexity of this problem is #P-hard because of the exponential growth of the number of solutions as the number of input polynomials increases. For example, ten polynomials of degree two may intersect in 1,024 isolated points (that is two to the power ten). Twenty quadratic polynomials may lead to 1,048,576 solutions (that is 1,024 times 1,024). So it is not too difficult to write down small input sequences that lead to a huge output.

Even as the computation of the total number of solutions may take a long time, numerical homotopy continuation methods have the advantage that they compute one solution after the other. A homotopy is a family of polynomial systems, connecting the system we want to solve with an easier to solve system, which is called the start system. Numerical continuation methods track the solution paths starting at known solutions of an easier system to the system we want to solve. We have an optimal homotopy if every path leads to a solution, that is: there are no divergent paths.

PHCpack offers optimal homotopies for systems that resemble linear-product structures, for geometric problems in enumerative geometry, and for sparse polynomial systems with sufficiently generic choices of the coefficients. While mathematically this sounds all good, most systems arising in practical applications have their own peculiar structure and so most homotopies will lead to diverging solution paths. In general, a polynomial system may have solution sets of many different dimensions, which renders the solving process challenging but a the same time still very interesting.

Version 1.0 of PHCpack was archived as Algorithm 795 by ACM Transactions on Mathematical Software. PHCpack is open source and free software which gives any user the same rights as in free speech. You can redistribute PHCpack and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; version 3 of the License.

1.2 Downloading and Installing

Executable versions of the program for various machine architectures and operating systems are available via <http://www.math.uic.edu/~jan/download.html>.

For the Windows operating systems, the executable version of phc is in the file `phc.exe` and is available for download in its uncompressed format. Using the plain version of phc on a Windows system requires the opening of a command prompt window.

For Mac OS X and Linux versions, the executable is tarred and gzipped. If the downloaded file is saved as `*phcv2_4p.tar.gz`, then the following commands to unzip and untar the downloaded file can be typed at the command prompt:

```
gunzip *phcv2_4p.tar.gz
tar xpf *phcv2_4p.tar
```

If all went well, typing `./phc` at the command prompt should bring up the welcome message and the screen with available options.

The executable `phc` gives access to almost all the functionality of PHCpack, including the multitasking capabilities for shared memory parallelism on multicore processors. For other parallel capabilities, such as distributed memory parallelism with the Message Passing Interface (MPI) and massive parallelism on a Graphics Processing Unit (GPU), one will need to compile the source code.

The source code is under version control at github, at <https://github.com/janverschelde/PHCpack>. To compile the source code, the gnu-ada compiler is needed. Free binary versions of the gnu-ada compiler are available at <http://libre.adacore.com>. The directory `Objects` in the source code provides makefiles for Linux, Mac OS X, and Windows operating systems.

When compiling from source, note that since version 2.4.35, the quad double library QDlib must be installed. On Linux systems, the qdlib.a must have been compiled with the -fPIC option for the shared object file for the C extension module of phcpy.

1.3 Project History

The software originated in the development of new homotopy algorithms to solve polynomial systems. The main novelty of the first release of the sources was the application of polyhedral homotopies in the blackbox solver. Polyhedral homotopies are generically optimal for sparse polynomial systems. Although the number of solutions may grow exponentially in the number of equations, variables, and degrees, for systems where the coefficients are sufficiently generic, every solution path defined by a polyhedral homotopy will lead to one isolated solution.

Version 2.0 of the code implemented SAGBI and Pieri homotopies to solve problem in enumerative geometry. A classical problem in Schubert calculus is the problem of the two lines that meet four general lines in 3-space. Pieri homotopies are generically optimal to compute all solutions to such geometric problems. They solve the output pole placement problem in linear systems control. With message passing, parallel versions of the Pieri homotopies lead to good speedups on parallel distributed memory computers.

Starting with version 2.0 was the gradual introduction of new homotopies to deal with positive dimensional solution sets. Cascades of homotopies provide generic points on every solution set, at every dimension. After the application of cascade homotopies to compute generic points on all equidimensional components, the application of monodromy loops with the linear trace stop test classifies the generic points on the equidimensional component into irreducible components. This leads to a numerical irreducible decomposition of the solution set of a polynomial system. Cascade of homotopies are the top down method. A bottom up method applies diagonal homotopies to intersect positive dimensional solution sets in an equation-by-equation solver.

To deal with singular solutions of polynomial systems, the deflation method was added in version 2.3. Version 2.3 was quickly followed by a bug release 2.3.01 and subsequently by many more quick releases. The introduction of the fast mixed volume calculator MixedVol in 2.3.13 was followed by capabilities to compute stable mixed volumes in 2.3.31, and an upgrade of the blackbox solver in version 2.3.34.

Shared memory multitasking provided the option -t, followed by the number of tasks, to speedup the path tracking. Our main motivation of parallelism is to offset the extra cost of multiprecision arithmetic, in particular double double and quad double arithmetic. Marking a milestone after one hundred quick releases, version 2.4 provided path tracking methods on graphics processing units. A collection of Python scripts defines a simple web interface to the blackbox solver and the path trackers, enabling the solution of polynomial systems in the cloud.

1.4 phcpy: An Application Programming Interface to PHCpack

Because code development on PHCpack has taken a very long time, looking at the code may be a bit too overwhelming at first. A good starting point could be the Python interface and in particular phcpy, with documentation at <http://www.math.uic.edu/~jan/phcpy_doc_html/index.html>.

The main executable `phc` built by the code in PHCpack is called at the command line with options to invoke specific tools and with file names as arguments in which the input and output data goes. In contrast, the scripting interface replaces the files with persistent objects and instead of selecting options from menus, the user runs scripts.

1.5 References

PHCpack relies for its fast mixed volume computation on MixedVol and on QDlib for its double double and quad double arithmetic. Pointers to the literature are mentioned below.

1. N. Bliss, J. Sommars, J. Verschelde and X. Yu: **Solving polynomial systems in the cloud with polynomial homotopy continuation.** In *Computer Algebra in Scientific Computing, 17th International Workshop, CASC 2015, Aachen, Germany*, edited by V.P. Gerdt, W. Koepf, E.W. Mayr, and E.V. Vorozhtsov. Volume 9301 of *Lecture Notes in Computer Science*, pages 87-100, Springer-Verlag, 2015.

2. T. Gao, T. Y. Li, M. Wu: **Algorithm 846: MixedVol: a software package for mixed-volume computation.** *ACM Transactions on Mathematical Software*, 31(4):555-560, 2005.

3. E. Gross, S. Petrovic, and J. Verschelde: **PHCpack in Macaulay2.** *The Journal of Software for Algebra and Geometry: Macaulay2*, 5:20-25, 2013.

4. Y. Guan and J. Verschelde: **PHClab: A MATLAB/Octave interface to PHCpack.** In *IMA Volume 148: Software for Algebraic Geometry*, edited by M. E. Stillman, N. Takayama, and J. Verschelde, pages 15-32, Springer-Verlag, 2008.

5. Y. Hida, X.S. Li, and D.H. Bailey: **Algorithms for quad-double precision floating point arithmetic.** In *15th IEEE Symposium on Computer Arithmetic (Arith-15 2001)*, 11-17 June 2001, Vail, CO, USA, pages 155-162. IEEE Computer Society, 2001. Shortened version of Technical Report LBNL-46996.

6. A. Leykin and J. Verschelde: **PHCmaple: A Maple Interface to the Numerical Homotopy Algorithms in PHCpack.** In the *Proceedings of the Tenth International Conference on Applications of Computer Algebra (ACA'2004)*, edited by Q. N. Tran, pages 139-147, 2004.

7. A. Leykin and J. Verschelde: **Interfacing with the Numerical Homotopy Algorithms in PHCpack.** In *proceedings of ICMS 2006, LNCS 4151*, edited by A. Iglesias and N. Takayama, pages 354-360, Springer-Verlag, 2006.

8. M. Lu. and B. He and Q. Luo **Supporting extended precision on graphics processors.** In *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana*, edited by A. Ailamaki and P.A. Boncz, pages 19-26, 2010.

9. K. Piret and J. Verschelde: **Sweeping Algebraic Curves for Singular Solutions.** *Journal of Computational and Applied Mathematics*, 234(4): 1228-1237, 2010.

10. A. J. Sommese, J. Verschelde, and C. W. Wampler. **Numerical irreducible decomposition using PHCpack.** In *Algebra, Geometry, and Software Systems*, edited by M. Joswig and N. Takayama, pages 109-130. Springer-Verlag, 2003.

11. J. Verschelde: **Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation.** *ACM Transactions on Mathematical Software*, 25(2):251–276, 1999.

12. J. Verschelde: **Polynomial homotopy continuation with PHCpack.** *ACM Communications in Computer Algebra*, 44(4):217-220, 2010.

13. J. Verschelde: **Modernizing PHCpack through phcpy.** In Proceedings of the 6th European Conference on Python in Science (EuroSciPy 2013), edited by Pierre de Buyl and Nelle Varoquaux, pages 71-76, 2014, available at <http://arxiv.org/abs/1310.0056>.

14. J. Verschelde and G. Yoffe. **Polynomial homotopies on multicore workstations.** In M.M. Maza and J.-L. Roch, editors, *Proceedings of the 4th International Workshop on Parallel Symbolic Computation (PASCO 2010), July 21-23 2010, Grenoble, France*, pages 131–140. ACM, 2010.

15. J. Verschelde and X. Yu: **Polynomial Homotopy Continuation on GPUs.** *ACM Communications in Computer Algebra*, 49(4):130-133, 2015.

1.6 Users

To demonstrate the relevance of the software, the first version of the software was released with a collection of about eighty different polynomial systems, collected from the literature. This section points to a different collection of problems, problems that have been solved by users of the software, without intervention of its developers.

The papers listed below report the use of PHCpack in the fields of algebraic statistics, communication networks, geometric constraint solving, real algebraic geometry, computation of Nash equilibria, signal processing, magnetism, mechanical design, computational geometry, computer vision, optimal control, image processing, pattern recognition, global optimization, and computational physics:

1. M. Abdullahi, B.I. Mshelia, and S. Hamma: **Solution of polynomial system using PHCpack**. *Journal of Physical Sciences and Innovation*, 4:44-53, 2012.

2. Min-Ho Ahn, Dong-Oh Nam and Chung-Nim Lee: **Self-Calibration with Varying Focal Lengths Using the Infinity Homography**. In *Proceedings of the 4th Asian Conference on Computer Vision* (ACCV2000), pages 140-145, 2000.

3. Gianni Amisano and Oreste Tristani: **Exact likelihood computation for nonlinear DSGE models with heteroskedastic innovations**. *Journal of Economic Dynamics and Control* 35:2167-2185, 2011.

4. D. Arzelier, C. Louembet, A. Rondepierre, and M. Kara-Zaitri: **A New Mixed Iterative Algorithm to Solve the Fuel-Optimal Linear Impulsive Rendezvous Problem.** *Journal of Optimization Theory and Applications*, 2013.

5. Bassi, I.G., Abdullahi Mohammed, and Okechukwu C.E.: **Analysis Of Solving Polynomial Equations Using Homotopy Continuation Method**. *International Journal of Engineering Research & Technology (IJERT)* 2(8):1401-1411, 2013.

6. Dmitry Batenkov: **Accurate solution of near-colliding Prony systems via decimation and homotopy continuation**. *Theoretical Computer Science* 681:1-232, 2017.

7. Daniel J. Bates and Frank Sottile: **Khovanskii-Rolle Continuation for Real Solutions**. *Foundations of Computational Mathematics* 11:563-587, 2011.

8. Jahan Bayat and Carl D. Crane III: **Closed-Form Equilibrium Analysis of Planar Tensegrity Mechanisms**. In *2006 Florida Conference on Recent Advances in Robotics*, FCRAR 2006.

9. Genevieve Belanger, Kristjan Kannike, Alexander Pukhov, and Martti Raidal: **Minimal semi-annihilating Z_n scalar dark matter**. *Journal of Cosmology and Astroparticle Physics*, June 2014 (Open Access).

10. Ivo W.M. Bleylevens, Michiel E. Hostenbach, and Ralf L.M. Peeters: **Polynomial Optimization and a Jacobi-Davidson type method for commuting matrices**, *Applied Mathematics and Computation* 224(1): 564-580, 2013.

11. Guy Bresler, Dustin Cartwright, David Tse: **Feasibility of Interference Alignment for the MIMO interference channel**. *IEEE Transactions on Information Theory* 60(9):5573-5586, 2014.

12. M.-L. G. Buot and D. St. P. Richards: **Counting and Locating the Solutions of Polynomial Systems of Maximum Likelihood Equations I**. *Journal of Symbolic Computation* 41(2): 234-244, 2005.

13. Max-Louis G. Buot, Serkan Hosten and Donald St. P. Richards: **Counting and locating the solutions of polynomial systems of maximum likelihood equations, II: The Behrens-Fisher problem**. *Statistica Sinica* 17(4):1343-1354, 2007.

14. Enric Celaya, Tom Creemers, Lluis Ros: **Exact interval propagation for the efficient solution of position analysis problems on planar linkages**. *Mechanism and Machine Theory* 54: 116-131, 2012.

15. Zachary Charles and Nigel Boston: **Exploiting algebraic structure in global optimization and the Belgian chocolate problem**. arXiv:1708.08114 [math.OC] 27 Aug 2017.

16. Tom Creemers, Josep M. Porta, Lluis Ros, and Federico Thomas: **Fast Multiresolutive Approximations of Planar Linkage Configuration Spaces**. *IEEE 2006 International Conference on Robotics and Automation.*

17. Marc Culler and Nathan M. Dunfield: **Orderability and Dehn filling.** arXiv:1602.03793v2 [math.GT] 17 June 2016.

18. R.S. Datta: **Using Computer Algebra To Compute Nash Equilibria**. In *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation (ISSAC 2003)*, pages 74-79, ACM 2003.

19. R.S. Datta: **Finding all Nash equilibria of a finite game using polynomial algebra**. *Economic Theory* 42(1):55-96, 2009.

20. B.H. Dayton: **Numerical Local Rings and Local Solution of Nonlinear Systems**. In *Proceedings of the 2007 International Workshop on Symbolic-Numeric Computation (SNC'07)*, pages 79-86, ACM 2007.

21. Max Demenkov: **Estimating region of attraction for polynomial vector fields by homotopy methods**. *ACM Communications in Computer Algebra* 46(3):84-85, 2012.

22. Max Demenkov: A Matlab Tool for Regions of Attraction Estimation via Numerical Algebraic Geometry.</B> In the *2015 International Conference on Mechanics - Seventh Polyakhov's Reading*, February 2-6, 2015, Russia, Saint Petersburg State University, Proceedings Edited by A.A. Tikhonov. IEEE 2015.

23. Ian H. Dinwoodie, Emily Gamundi, and Ed Mosteig: **Multiple Solutions for Blocking Probabilities in Asymmetric Networks**. *Open Systems and Information Dynamics* 12(3):273-288, 2005.

24. Csaba Domokos and Zoltan Kato: **Parametric Estimation of Affine Deformations of Planar Shapes**. *Pattern Recognition*, 2009. In press.

25. C. Durand and C.M. Hoffmann: **Variational Constraints in 3D**. In *Proceedings of the International Conference on Shape Modeling and Applications*, Aizu-Wakamatsu, Japan, pages 90-98, IEEE Computer Society, 1999.

26. C. Durand and C.M. Hoffmann: **A systematic framework for solving geometric constraints analytically**. *Journal of Symbolic Computation* 30(5):493-520, 2000.

27. I.Z. Emiris, E. Tsigaridas, G. Tzoumas: **The predicates for the Voronoi diagram of ellipses**. In *Proc. ACM Symp. Comput. Geom.* 2006.

28. Jonathan P. Epperlein and Bassam Bamieh: **A Frequency Domain Method for Optimal Periodic Control**. 2012 American Control Conference (ACC), pages 5501-5506, IEEE 2012.

29. F. Ferrari: **On the geometry of super Yang-Mills theories: phases and irreducible polynomials**. *Journal of High Energy Physics* 1, paper 26, 2009.

30. Jaime Gallardo-Alvarado: **A simple method to solve the forward displacement analysis of the general six-legged parallel manipulator**. *Robotics and Computer-Integrated Manufacturing* 30:55-61, 2014.

31. Jaime Gallardo-Alvarado: **Gough's Tyre Testing Machine**. Chapter 12 of *Kinematic Analysis of Parallel Manipulators by Algebraic Screw Theory*, pages 255-280, Springer-Verslag, 2016.

32. Jaime Gallardo-Alvarado and Juan-de-Dios Posadas-Garcia: **Mobility analysis and kinematics of the semi-general 2(3-RPS) series-parallel manipulator**. *Robotics and Computer-Integrated Manufactoring* 29(6): 463-472, 2013.

33. Jaime Gallardo-Alvarado, Mohammad H. Abedinnasab, and Daniel Lichtblau: **Simplified Kinematics for a Parallel Manipulator Generator of the Schoenflies Motion**. *Journal of Mechanisms and Robotics* 8(6):061020-061020-10, 2016.

34. Bertrand Haas: **A Simple Counterexample to Kouchnirenko's Conjecture**. *Beitraege zur Algebra und Geometrie/Contributions to Algebra and Geometry* 43(1):1-8, 2002.

35. Adlane Habed and Boubakeur Boufama: **Camera self-calibration from bivariate polynomial equations and the coplanarity constraint**. *Image and Vision Computing* 24(5):498-514, 2006.

36. Marshall Hampton and Richard Moeckel: **Finiteness of stationary configurations of the four-vortex problem**. *Transactions of the American Mathematical Society* 361(3): 1317-1332, 2009.

37. Jonathan Hauenstein, Jose Israel Rodriguez, and Bernd Sturmfels: **Maximum Likelihood for Matrices with Rank Constraints**. *Journal of Algebraic Statistics* 5(1): 18-38, 2014.

38. Christoph Hellings, David A. Schmidt, and Wolfgang Utschick: **Optimized beamforming for the two stream MIMO interference channel at high SNR**. In 2009 Internatial ITG Workshop on Smart Antennas (WSA 2009), February 16-19, Berlin, Germany, pages 88-95.

39. Gabor Horvath: **Moment Matching-Based Distribution Fitting with Generalized Hyper-Erlang Distributions**. In *Analytical and Stochastic Modeling Techniques and Applications*, Lecture Notes in Computer Science, Volume 7984, pages 232-246, 2013.

40. X.G. Huang: **Forward Kinematics for a Parallel Platform Robot**. *Communications in Computer and Information Sciences* 86:529-532, 2011.

41. Xiguang Huang, Qizheng Liao, Shimin Wei, and Qiang Xu: **Five precision point-path synthesis of planar four-bar linkage using algebraic method**. *Frontiers of Electrical and Electronic Engineering in China* 3(4):470-474, 2008.

42. Xiguang Huang, Qizheng Liao, Shimin Wei, Qiang Xu, and Shuguang Huang: **The 4SPS-2CCS generalized Stewart-Gough Platform mechanisms and its direct kinematics**. In *Proceedings of the 2007 IEEE International Conference on Mechatronics and Automation*, August 5-8, 2007, Harbin, China. Pages 2472-2477, 2007.

43. Libin Jiao, Bo Dong, Jintao Zhang, and Bo Yu: **Polynomial Homotopy Methods for the Sparse Interpolation Problem Part I: Equally Spaced Sampling**. *SIAM J. Numer. Anal.* 54(1): 462-480, 2016.

44. Hamadi Jamali, Tokunbo Ogunfunmi: **Stationary points of the finite length constant modulus optimization**. *Signal Processing* 82(4): 625-641, 2002.

45. A. Jensen, A. Leykin, and J. Yu: **Computing tropical curves via homotopy continuation.** *Experimental Mathematics* 25(1): 83–93, 2016.

46. Bjorn Johansson, Magnus Oskarsson, and Kalle Astrom: **Structure and motion estimation from complex features in three views**. In the Online ICVGIP-2002 Proceedings (Indian Conference on Computer Vision, Graphics and Image Processing).

47. M. Kara-Zaitri, D. Arzelier, and C. Louembet: **Mixed iterative algorithm for solving optimal implusive time-fixed rendezvous problem**. *American Institute of Aeronautics and Astronautics Guidance, Navigation, and Control Conference*, Toronto, Canada, 02-05 August 2010.

48. P.U. Lamalle, A. Messiaen, P. Dumortier, F. Durodie, M. Evrard, F. Louche: **Study of mutual coupling effects in the antenna array of the ICRH plug-in for ITER**. *Fusion Engineering and Design* 74:359-365, 2005.

49. E. Lee and C. Mavroidis: **Solving the Geometric Design Problem of Spatial 3R Robot Manipulators Using Polynomial Continuation**. *Journal of Mechanical Design, Transactions of the ASME* 124(4):652-661, 2002.

50. E. Lee and C. Mavroidis: **Four Precision Points Geometric Design of Spatial 3R Manipulators**. In the *Proceedings of the 11th World Congress in Mechanism and Machine Sciences*, August 18-21, 2003, Tianjin, China. China Machinery Press, edited by Tian Huang.

51. E. Lee and C. Mavroidis: **Geometric Design of 3R Manipulators for Reaching Four End-Effector Spatial Poses**. *International Journal for Robotics Research*, 23(3):247-254, 2004.

52. E. Lee, C. Mavroidis, and J. Morman: **Geometric Design of Spatial 3R Manipulators**. In *Proceedings of the 2002 NSF Design, Service, and Manufacturing Grantees and Research Conference*, San Juan, Puerto Rico, January 7-10, 2002.

53. Dimitri Leggas and Oleg V. Tsodikov: **Determination of small crystal structures from a minimum set of diffraction intensities by homotopy continuation**. *Acta Crystallographica Section A* 71(3): 319-324, 2015.

54. Dawei Leng and Weidong Sun: **Finding All the Solutions of PnP Problem**. In *IST 2009 - International Workshop on Imaging Systems and Techniques*, Shenzhen, China, May 11-12, 2009. Pages 348-352, IEEE, 2009.

55. Anton Leykin: **Numerical Primary Decomposition**. In *Proceedings of ISSAC 2008*, edited by David Jeffrey, pages 165-164, ACM 2008.

56. Anton Leykin and Frank Sottile: **Computing Monodromy via Parallel Homotopy Continuation**. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation (PASCO'07)*, pages 97-98, ACM 2007. (on CDROM)

57. Anton Leykin and Frank Sottile: **Galois groups of Schubert problems via homotopy computation**. *Mathematics of Computation* 78: 1749-1765, 2009.

58. Shaobai Li, Srinandan Dasmahapatra, and Koushik Maharatna: **Dynamical System Approach for Edge Detection Using Coupled FitzHugh-Naguma Neurons**. *IEEE Transactions on Image Processing* 24(12), 5206-5219, 2015.

59. Ross A. Lippert: **Fixing multiple eigenvalues by a minimal perturbation**. *Linear Algebra Appl.* 432(7): 1785-1817, 2010.

60. M. Maniatis and O. Nachtmann: **Stability and symmetry breaking in the general three-Higgs-double model**. *Journal of High Energy Physics* 2015:58, February 2015.

61. Hyosang Moon and Nina P. Robson: **Design of spatial non-anthropomorphic articulated systems based on arm joint constraint kinematic data for human interactive robotics applications**. DETC2015-46530. In the *Proceedings of the ASME 2015 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*. IDETC/CIE 2015. August 2-5, 2015, Boston Massachusetts.

62. Marc Moreno Maza, Greg Reid, Robin Scott, and Wenyuan Wu: **On Approximate Triangular Decompositions I. Dimension Zero**. In the *SNC 2005 Proceedings*. International Workshop on Symbolic-Numeric Computation. Xi'an, China, July 19-21, 2005. Edited by Dongming Wang and Lihong Zhi. Pages 250-275, 2005.

63. Andrew J. Newell: **Transition to supermagnetism in chains of magnetosome crystals**. *Geochemistry Geophysics Geosystems* 10(11):1-19, 2009.

64. M. Oskarsson, A. Zisserman and K. Astrom: **Minimal Projective Reconstruction for combinations of Points and Lines in Three Views**. In the *Electronic Proceedings of BMVC2002 - The 13th British Machine Vision Conference 2002*, pages 63 - 72.

65. P.A. Parrilo and B. Sturmfels. **Minimizing polynomial functions**. In S. Basu and L. Gonzalez-Vega, editors, *Algorithmic and quantitative real algebraic geometry*, volume 60 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 83-99. AMS, 2003.

66. Alba Perez and J.M. McCarthy: **Dual Quaternion Synthesis of Constrained Robotic Systems**. *Journal of Mechanical Design* 126(3): 425-435, 2004.

67. Nina Patarinsky-Robson, J. Michael McCarthy, and Irem Y. Tumer: **The algebraic synthesis of a spatial TS chain for a prescribed acceleration task**. *Mechanism and Machine Theory* 43(10): 1268-1280, 2008.

68. Nina Patarinsky-Robson, J. Michael McCarthy, and Irem Y. Tumer: **Failure Recovery Planning for an Arm Mounted on an Exploratory Rover**. *IEEE Transactions on Robotics* 25(6):1448-1453, 2009.

69. Jose Israel Rodriguez: **Combinatorial excess intersection**. *Journal of Symbolic Computation* 68(2): 297-307, 2015.

70. Roger E. Sanchez-Alonso, Jose-Joel Gonzalez-Barbosa, Eduardo Castilo-Castaneda, and Jaime Gallardo-Alvarado: **Kinematic analysis of a novel 2(3-RUS) parallel manipulator**. *Robotica*, available on CJO2015.

71. H. Schreiber, K. Meer, and B.J. Schmitt: **Dimensional synthesis of planar Stephenson mechanisms for motion generation using circlepoint search and homotopy methods**. *Mechanism and Machine Theory* 37(7):717-737, 2002.

72. Ben Shirt-Ediss, Ricard V. Sole, and Kepa Ruiz-Mirazo: **Emergent Chemical Behavior in Variable-Volume Protocells**. *Life* 5: 181-121, 2015.

73. Hythem Sidky, Jonathan K. Whitmer, and Dhagash Mehta: **Reliable mixture critical point computation using polynomial homotopy continuation**. *AIChE Journal. Thermodynamics and Molecular-Scale Phenomena*, 2016. doi:10.1002/aic.15319

74. Frank Sottile: **Real Schubert Calculus: Polynomial systems and a conjecture of Shapiro and Shapiro**. *Experimental Mathematics* 9(2): 161-182, 2000.

75. H. Stewenius and K. Astrom: **Structure and Motion Problems for Multiple Rigidly Moving Cameras**. In *Computer Vision - ECCV 2004: 8th European Conference on Computer Vision, Prague, Czech Republic, May 11-14, 2004. Proceedings, Part III*. Edited by T. Pajdla and J. Matas. Lecture Notes in Computer Science 3023, pages 252-263, Springer, 2004.

76. H.-J. Su and J.M. McCarthy: **Kinematic Synthesis of RPS Serial Chains**. In the *Proceedings of the ASME Design Engineering Technical Conferences* (CDROM). Paper DETC03/DAC-48813. Chicago, IL, Sept. 02-06, 2003.

77. H.-J. Su and J.M. McCarthy: **Synthesis of Compliant Mechanisms with Specified Equilibrium Positions**. In the *Proceedings of the ASME International Design Engineering Technical Conferences*. Paper DETC 2005-85085. Long Beach, CA, Sept. 24-28 2005.

78. H.-J. Su and J.M. McCarthy: **Kinematic Synthesis of RPS Serial Chains for a Given Set of Task Positions**. *Mechanism and Machine Theory* 40(7):757-775, 2005

79. H.-J. Su and J.M. McCarthy: **A Polynomial Homotopy Formulation of the Inverse Static Analysis of Planar Compliant Mechanisms**. *ASME Journal of Mechanical Design* 128(4): 776-786, 2006.

80. H.-J. Su, C.W. Wampler, and J.M. McCarthy: **Geometric Design of Cylindric PRS Serial Chains**. *ASME Design Engineering Technical Conferences*, Chicago, IL, Sep 2-6, 2003.

81. Weronika J. Swiechowicz and Yuanfang Xiang: **Numerical Methods for Estimating Correlation Coefficient of Trivariate Gaussians** (sponsor: Sonja Petrovic) in Volume 8 of *SIAM Undergraduate Research Online (SIURO)*, 2015.

82. Attila Tanács and Joakim Lindblad and Nataša Sladoje and Zoltan Ka: **Estimation of linear deformations of 2D and 3D fuzzy objects**. *Pattern Recognition* 48(4):1391-1403, 2015.

83. N. Trawny, X.S. Zhou, K.X. Zhou, S.I. Roumeliotis: **3D Relative Pose Estimation from Distance-Only Measurements**. In the *Proceedings of the 2007/IEEE/RSJ International Conference on intelligent Robots and Systems*. San Diego, CA, Oct 29-Nov 2, 2007, pages 1071-1078, IEEE, 2007.

84. T. Turocy: **Towards a black-box solver for finite games: Computing all equilibria with Gambit and PHCpack**. In *Software for Algebraic Geometry*, volume 148 of the IMA volumes in Mathematics and its Applications, edited by M.E. Stillman, N. Takayama, and J. Verschelde, pages 133-148, Springer-Verlag, 2008.

85. Konstantin Usevich and Ivan Markovsky: **Structured low-rank approximation as a rational function minimization**. In 16th IFAC Symposium on System Identification Brussels, 11-13 Jul 2012, pages 722-727.

86. C.W. Wampler: **Isotropic coordinates, circularity and Bezout numbers: planar kinematics from a new perspective**. In the *Proceedings of the 1996 ASME Design Engineering Technical Conference*. Irvine, CA, Aug 18-22, 1996. Available on CD-ROM.

87. Wenyuan Wu and Greg Reid: **Symbolic-numeric computation of implicit Riquier bases for PDE**. In the *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, edited by C.W. Brown, pages 377-385, ACM 2007.

88. Jonathan Widger and Daniel Grosu: **Parallel Computation of Nash Equilibria in N-Player Games**. In the *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering (CSE 2009)*, August 29-31, 2009, Vancouver, Canada, pages 209-215.

89. F. Xie, G. Reid, and S. Valluri: **A numerical method for the one dimensional action functional for FBG structures**. *Can J. Phys.* 76: 1-21, 2002.

90. Hong Bing Xin, Qiang Huang, and Yueqing Yu: **Position and Orientation Analyses of Mechanism by PHCpack Solver of Homotopy Continuation**. *Applied Mechanics and Materials* 152-254: 1779-1784, 2012.

91. Ke-hu Yang, Dan-ying Lu, Xiao-qing Kuang, and Wen-Shen Yu: **Harmonic Elimination for Multilevel Converters with Unequal DC levels by Using the Polynomial Homotopy Continuation Algorithm**. In the Proceedings of the 35th Chinese Control Conference, July 27-29, 2016, Chengdu, China, pages 9969-9973, IEEE.

92. K. Yang and R. Orsi: **Static output feedback pole placement via a trust region approach**. *IEEE Transactions on Automatic Control* 52(11): 2146-2150, 2007.

93. Yan Yang, Yao Zhang, Fangxing Li, and Haoyong Chen: **Computing All Nash Equilibria of Multiplayer Games in Electricity Markets by Solving Polynomial Equations**. *IEEE Transactions on Power Systems* 27(1): 81-91, 2012.

94. Jun Zhang and Mohan Sarovar: **Identification of open quantum systems from observable time traces**. *Physical Review A* 91, 052121, 2015.

95. Shiqiang Zhang, Shufang Zhang, and Yan Wan: **Biorthogonal Wavelet Construction Using Homotopy Method**. *Chinese Journal of Electronics* 24(4), pages 772-775, 2015.

96. Xun S. Zhou and Stergios I. Roumeliotis: **Determining 3-D Relative Transformations for Any Combination of Range and Bearing Measurements.** *IEEE Transactions on Robotics* 29(2):458-474, 2013.

97. Lifeng Zhou, Hai-Jun Su, Alexander E. Marras, Chao-Min Huang, Carlos E. Castro: **Projection kinematic analysis of DNA origami mechanisms based on a two-dimensional TEM image.** Mechanisms and Machine Theory 109:22-38, 2017.

In addition to the publications listed above, PHCpack was used as a benchmark to measure the progress of new algorithms in the following papers:

98. Ali Baharev, Ferenc Domes, Arnold Neumaier: **A robust approach for finding all well-separated solutions of sparse systems of nonlinear equations**. *Numerical Algorithms*, pages 1-27, 2016 (online first).

99. Ada Boralevi, Jasper van Doornmalen, Jan Draisma, Michiel E. Hochstenbach, and Bor Plestenjak: **Uniform Determinantal Representations**. *SIAM J. Appl. Algebra Geometry*, vol. 1, pages 415-441, 2017.

100. Timothy Duff, Cvetelina Hill, Anders Jensen, Kisun Lee, Anton Leykin, and Jeff Sommars: **Solving polynomial systems via homotopy continuation and monodromy**. arXiv:1609.08722v2 [math.AG] 2 Nov 2016

101. T. Gao and T.Y. Li: **Mixed volume computation via linear programming**. *Taiwanese Journal of Mathematics* 4(4): 599-619, 2000.

102. T. Gao and T.Y. Li: **Mixed volume computation for semi-mixed systems**. *Discrete Comput. Geom.* 29(2):257-277, 2003.

103. L. Granvilliers: **On the Combination of Interval Constraint Solvers**. *Reliable Computing* 7(6): 467-483, 2001.

104. Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler: **Regeneration Homotopies for Solving Systems of Polynomials** *Mathematics of Computation* 80(273): 345-377, 2011.

105. S. Kim and M. Kojima: **Numerical Stability of Path Tracing in Polyhedral Homotopy Continuation Methods**. *Computing* 73(4): 329-348, 2004.

106. Y. Lebbah, C. Michel, M. Rueher, D. Daney, and J.P. Merlet: **Efficient and safe global constraints for handling numerical constraint systems**. *SIAM J. Numer. Anal.* 42(5):2076-2097, 2005.

107. T.L. Lee, T.Y. Li, and C.H. Tsai: **HOM4PS-2.0: a software package for solving polynomial systems by the polyhedral homotopy continuation method**. *Computing* 83(2-3): 109-133, 2008.

108. Anton Leykin: **Numerical Algebraic Geometry**. *The Journal of Software for Algebra and Geometry* volume 3, pages 5-10, 2011.

109. T.Y. Li and X. Li: **Finding Mixed Cells in the Mixed Volume Computation**. *Foundations of Computational Mathematics* 1(2): 161-181, 2001.

110. T.Y. Li, X. Wang, and M. Wu: **Numerical Schubert Calculus by the Pieri Homotopy Algorithm**. *SIAM J. Numer Anal.* 40(2): 578-600, 2002.

111. J.M. Porta, L. Ros, T. Creemers, and F. Thomas: **Box approximations of planar linkage configuration spaces**. *Journal of Mechanical Design* 129(4):397-405, 2007.

112. Laurent Sorber, Marc Van Barel, and Lieven De Lathauwer: **Numerical solution of bivariate and polyanalytic polynomial systems**. *SIAM J. Numer. Anal.* 52(4):1551-1572, 2014.

113. Yang Sun, Yu-Hui Tao, Feng-Shan Bai: **Incomplete Groebner basis as a preconditioner for polynomial systems**. *Journal of Computational and Applied Mathematics* 226(1):2-9, 2009.

PHCpack was used to develop new homotopy algorithms:

114. Bo Dong, Bo Yu, and Yan Yu: **A symmetric and hybrid polynomial system solving method for mixed trigonometric polynomial systems**. *Mathematics of Computation* 83(288): 1847-1868, 2014.

115. Bo Yu and Bo Dong: **A hybrid polynomial system solving method for mixed trigonometric polynomial systems**. *SIAM J. Numer. Anal.* 46(3): 1503-1518, 2008.

116. Xuping Zhang, Jintao Zhang, and Bo Yu: **Eigenfunction expansion method for multiple solutions of semilinear elliptic equations with polynomial nonlinearity** *SIAM J. Numer. Anal.* 51(5): 2680-2699, 2013.

Last, but certainly not least, there is the wonderful book of Bernd Sturmfels which contains a section on computing Nash equilibria with PHCpack.

117. B. Sturmfels: **Solving Systems of Polynomial Equations**. CBMS Regional Conference Series of the AMS, Number 97, 2002.

So we have to end quoting Bernd Sturmfels: *polynomial systems are for everyone.*

1.7 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. 9804846, 0105739, 0134611, 0410036, 0713018, 1115777, and 1440534. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Since 2001, the code in PHCpack improved thanks to the contributions of many PhD students at the University of Illinois at Chicago. Their names, titles of PhD dissertation, and year of PhD are listed below:

1. Yusong Wang: *Computing Dynamic Output Feedback Laws with Pieri Homotopies on a Parallel Computer*, 2005.
2. Ailing Zhao: *Newton's Method with Deflation for Isolated Singularities of Polynomial Systems*, 2007.
3. Yan Zhuang: *Parallel Implementation of Polyhedral Homotopy Methods*, 2007.
4. Kathy Piret: *Computing Critical Points of Polynomial Systems using PHCpack and Python*, 2008.
5. Yun Guan: *Numerical Homotopies for Algebraic Sets on a Parallel Computer*, 2010.
6. Genady Yoffe: *Using Parallelism to compensate for Extended Precision in Path Tracking for Polynomial System Solving*, 2012.
7. Danko Adrovic: *Solving Polynomial Systems with Tropical Methods*, 2012.
8. Xiangcheng Yu: *Accelerating Polynomial Homotopy Continuation on Graphics Processing Units*, 2015.

Anton Leykin contributed to the application of message passing in a parallel implementation of monodromy to decompose an equidimensional solution set into irreducible components. The Maple interface `PHCmaple` was written jointly with Anton Leykin. The work of Anton Leykin also paved the way for the Macaulay2 interface, which was further developed into `PHCpack.m2` in joint work with Elizabeth Gross and Sonja Petrovic. The `PHCpack.m2` (and also PHCpack itself) improved during various Macaulay2 workshops, with the help of Taylor Brysiewicz, Diego Cifuentes, Corey Harris, Kaie Kubjas, Anne Seigal, and Jeff Sommars.

The software has been developed with GNAT GPL, the gnu-ada compiler.

CHAPTER 2

Tutorial

This chapter provides a short tutorial, mainly for use at the command line. Interfaces for Maple, MATLAB (Octave), SageMath, and Python provide a scripting environment.

2.1 Input formats

A lot of examples are contained in the database of Demo systems, which can be downloaded in zipped and tarred format from the above web site.

The input file starts with the number of equations and (optionally, but necessary in case of an unequal number) the number of unknowns. For example, the polynomial system of Bertrand Haas (which provided a counterexample for the conjecture of Koushnirenko) is represented as follows

```
2
  x**108 + 1.1*y**54 - 1.1*y;
  y**108 + 1.1*x**54 - 1.1*x;
```

For future use, we save this system in the file `haas`. Observe that every polynomial terminates with a semicolon. The exponentiation may also be denoted by a hat instead of a double asterix.

The forbidden symbols to denote names of variables are `i` and `I`, because they both represent the square root of -1. Also forbidden are `e` and `E` because they are used in the scientific notation of floating-point numbers, like 0.01 = 1.0e-2 = 1.0E-2.

The equations defining the adjacent 2-by-2 minors of a general 2-by-4 matrix are represented as

```
3 8
 x11*x22 - x21*x12;
 x12*x23 - x22*x13;
 x13*x24 - x23*x14;
```

thus as 3 polynomials in the 8 undeterminates of a general 2-by-4 matrix. We save this file as `adjmin4`.

The program also accepts polynomials in factored form, for example,

```
5
 (a-1)*(b-5)*(c-9)*(d-13) - 21;
 (a-2)*(b-6)*(c-10)*(f-17) - 22;
 (a-3)*(b-7)*(d-14)*(f-18) - 23;
 (a-4)*(c-11)*(d-15)*(f-19) - 24;
 (b-8)*(c-12)*(d-16)*(f-20) - 25;
```

is a valid input file for phc. Note that we replaced the logical e variable by f. We save this input in the file with name `multilin`.

2.2 Interfaces

The software is developed for command line interactions. Because there is no interpreter provided with PHCpack, there are interfaces to computer algebra systems such as for example Maple.

From the web site mentioned above we can download the Maple procedure run_phc and an example worksheet on how to use this procedure. The Maple procedure requires only two arguments: the path name ending in the name of the executable version of the program, and a list of polynomials. This procedure sets up the input file for phc, calls the blackbox solver and returns the list of approximate solutions. This list is returned in Maple format.

Other interfaces are PHClab (for Octave and MATLAB), phc.py (for SageMath), and PHCpack.m2 (for Macaulay 2). These interfaces require only the executable phc to be present in some directory contained in the execution path. Interfaces for C and C++ programmers require the compilation of the source code. For Python, a shared object file needs to exist for the particular architecture.

A diagram of the interfaces to PHCpack and phc is depicted in Fig. 2.1.

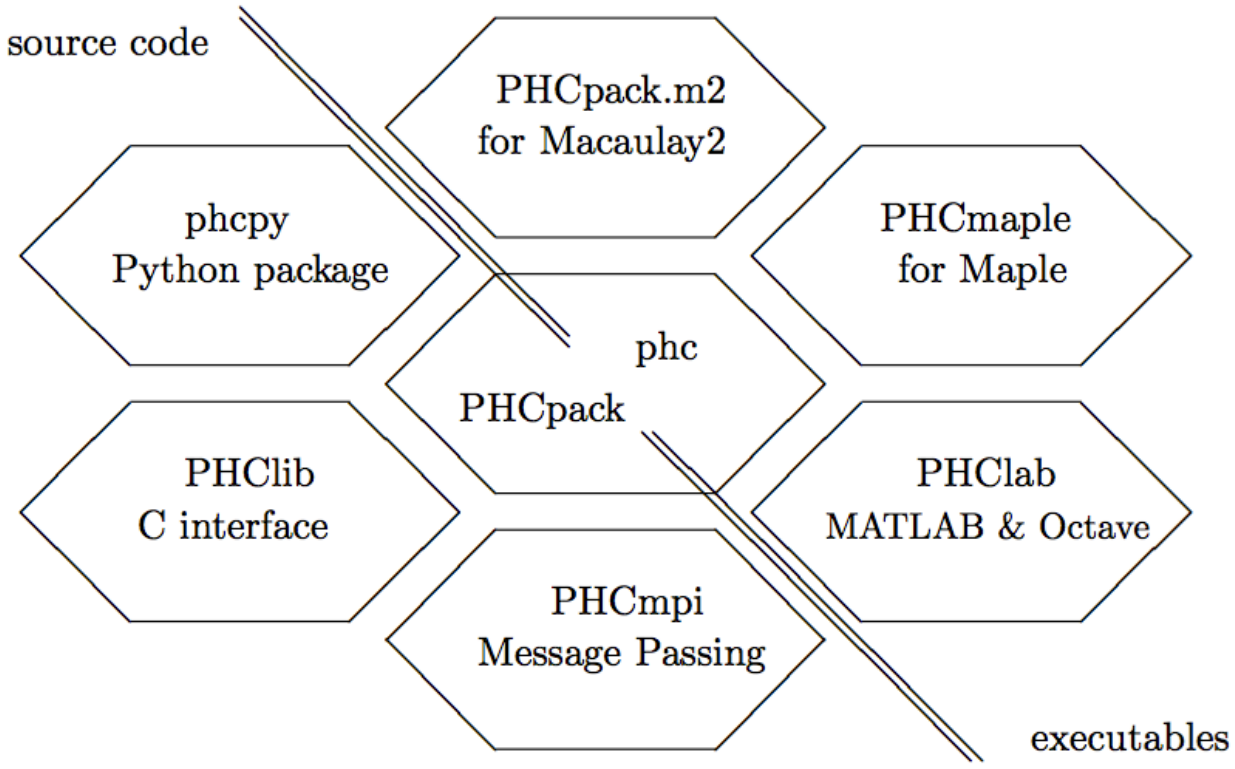

Fig. 2.1: Interfaces either require the source code or only the executable.

The interfaces PHCpack.m2, PHCmaple, PHClab, shown to the right of the antidiagonal require only the executable version phc. The other interfaces PHClib, PHCmpi, and phcpy are tied to the source code.

The phc.py is an optional package, available in the distribution of SageMath. Another, perhaps more natural interface to SageMath, is to extend the Python interpreter of SageMath with phcpy.

2.3 Calling the blackbox solver

The blackbox solver works reasonably well to approximate all isolated solutions of a polynomial system. On the system we saved earlier in the file multilin, we invoke the blackbox solver typing at the command prompt

```
/tmp/phc -b multilin multilin.phc
```

The output of the solver will be sent to the file multilin.phc. In case the input file did not yet contain any solutions, the solution list will be appended to the input file.

We now explain the format of the solutions, for example, the last solution in the list occurs in the following format:

```
solution 44 :    start residual :  1.887E-14   #iterations : 1   success
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 a :  5.50304308029581E+00  -6.13068078142107E-44
 b :  8.32523889626848E+00  -5.18918337570284E-45
 c :  1.01021324864917E+01  -1.29182202179944E-45
 d :  1.42724963260133E+01   1.38159270467025E-44
 f :  4.34451307203401E+01  -6.26380413553193E-43
== err :  3.829E-12 = rco :  3.749E-03 = res :  2.730E-14 = real regular ==
```

This is the actual output of the root refiner. As the residual at the end of the solution path and at the start of the root refinement is already 1.887E-14, one iteration of Newton's method suffices to confirm the quality of the root.

The next line in the output indicates that we reached the end of the path, at t=1, properly. The multiplicity of the root is one, as indicated by m = 1. Then we see the values for the five variables, as pairs of two floating-point numbers: the real and imaginary part of each value. The last line summarizes the numerical quality of the root. The value for err is the magnitude of the last correction term used in Newton's method. The number for rco is an estimate for the inverse condition number of the root. Here this means that we are guaranteed to have all decimal places correct, except for the last three decimal places. The last number represents the residual, the magnitude of the vector evaluated at the root.

2.4 Running the program in full mode

If we just type in /tmp/phc without any option, we run the program in full mode and will pass through all the main menus. A nice application is the verification of the counterexample of Bertrand Haas. We type in haas when the program asks us for the name of the input file. As the output may be rather large, we better save the output file on /tmp. As we run through all the menus, for this system, a good choice is given by the default, so we can type in 0 to answer every question. At the very end, for the output format, it may be good to type in 1 instead of 0, so we can see the progress of the program as it adds solution after solution to the output file.

If we look at the output file for the system in `multilin`, then we see that the mixed volume equals the 4-homogeneous Bezout number. Since polyhedral methods (e.g. to compute the mixed volume) are computationally more expensive than the solvers based on product homotopies, we can solve the same problem faster. If we run the program on the system in multilin in full mode, we can construct a multi-homogeneous homotopy as follows. At the menu for Root Counts and Method to Construct Start Systems, we type in 1 to select a multi-homogeneous Bezout number. Since there are only 52 possible partitions of a set of four unknowns, it does not take that long for the program to try all 52

partitions and to retain that partition that yields the lowest Bezout number. Once we have this partition, we leave the root counting menu with 0, and construct a linear-product system typing 2 in the menu to construct m-homogeneous start systems. We can save the start system in the file multilin_start (only used for backup). Now we continue just as before.

2.5 Running the program in toolbox mode

Skipping the preconditioning stage (scaling and reduction), we can compute root counts and construct start systems via the option `-r`, thus calling the program as `phc -r`. One important submenu is the mixed-volume computation, invoked via `phc -m`.

Once we created an appropriate start system, we can call the path trackers via the option `-p`. Calling the program as `phc -p` is useful if we have to solve a slightly modified problem. For instance, suppose we change the coefficients of the system in multilin, then we can still use multilin_start to solve the system with modified coefficients, using the `-p` option. In this way we use a cheater's homotopy, performing a kind of coefficient-parameter polynomial continuation.

2.6 Dealing with components of solutions

Consider the system of adjacent minors, we previously saved as `adjmin4`. We first must construct a suitable embedding to get to a system with as many equations as unknowns. We call `phc -c` and type 5 as top dimension. The system the program produces is saved as `adjmin4e5`. The blackbox solver has no difficulty to solve this problem and appends the witness points to the file `adjmin4e5`. To compute the irreducible decomposition, we may use the monodromy breakup algorithm, selecting 2 from the menu that comes up when we can the program with the option `-f`.

CHAPTER 3

User Manual

This chapter describes all options of the executable `phc`.

3.1 Text Options of the Executable phc

Text options of `phc` do not compute anything, but provide text information about `phc`.

While the regular options of `phc` start with a single dash `-`, the text options are double dashed, they start with `--`.

3.1.1 phc –version : displays the current version string

Typing at the command prompt `phc --version` displays the version string which includes the current version number and the release date.

3.1.2 phc –license : writes the license to screen

The output of

```
phc --license
```

is

```
PHCpack is free and open source software.
You can redistribute the code and/or modify it under
the GNU General Pulic License as published by
the Free Software Foundation.
```

3.1.3 phc –cite : how to cite PHCpack

Typing `phc --cite` at the command prompt displays

```
To cite PHCpack in publications use:

Jan Verschelde:
Algorithm 795: PHCpack: A general-purpose solver for polynomial
systems by homotopy continuation.
ACM Transactions on Mathematical Software, 25(2):251--276, 1999.
```

3.1.4 phc –help : writes helpful information to screen

Typing at the command prompt `phc --help` provides some help to get started with the quickest use, that is: with the blackbox solver.

To obtain help about the blackbox solver, type `phc -b --help` or `phc --help -b` where the `--help` may be abbreviated by `-h`.

3.2 Options of the Executable phc

For many small to moderate size problems, the most convenient way to get an answer of phc is to call the blackbox solver with the option -b.

3.2.1 phc -0 : random numbers with fixed seed for repeatable runs

Many homotopy algorithms generate random constants. With each run, the current time is used to generate another seed for the random number generator, leading to different random constants in each run. As different random values give different random start systems, this may cause differences in the solution paths and fluctuations in the executation time. Another notable effect of generating a different random constant each time is that the order of the solutions in the list may differ. Although the same solutions should be found with each run, a solution that appears first in one run may turn out last in another run.

With the option -0, a fixed seed is used in each run. This option can be combined with the blackbox solver (phc -b), e.g.: phc -b -0 or phc -0 -b.

Since version 2.3.89, the option -0 is extended so the user may give the digits of the seed to be used. For example, calling phc as phc -089348224 will initialize the random number generator with the seed 89348224. Just calling phc as phc -0 will still result in using the same fixed seed as before in each run.

To reproduce a run with any seed (when the option -0 was not used), we can look at the output file, for the line

```
Seed used in random number generators : 407.
```

which appears at the end of the output of phc -b. Running phc -b -0407 on the same input file as before will generate the same sequences of random numbers and thus the same output.

A homotopy is a family of polynomial systems. In an artificial parameter homotopy there is one parameter t usually going from 0 to 1. If we want to solve the system $f(\mathbf{x}) = \mathbf{0}$ and we have a system $g(\mathbf{x}) = \mathbf{0}$ then a typical homotopy is $h(\mathbf{x}, t) = \mathbf{0}$ as below.

$$h(\mathbf{x}, t) = \gamma(1-t)g(\mathbf{x}) + tf(\mathbf{x}) = \mathbf{0}.$$

The γ is a complex constant on the unit circle in the complex plane, generated uniformly at random. If all solutions of $g(\mathbf{x}) = \mathbf{0}$ are isolated and of multiplicity one, then only for a finite number of complex values for t the homotopy $h(\mathbf{x}, t) = \mathbf{0}$ has singular solutions. But since we consider $t \in [0, 1]$ and since the values for t for which $h(\mathbf{x}, t) = \mathbf{0}$ are complex, the interval $[0, 1($ will with probability one not contain any of the bad complex values for t and therefore no singular solutions for $t \in [0, 1($ will occur.

Note that, for this gamma trick to work, the order of the operations matters. We first give the program the system $f(\mathbf{x}) = \mathbf{0}$ and then either also give the system $g(\mathbf{x}) = \mathbf{0}$ or let the program generate a suitable $g(\mathbf{x}) = \mathbf{0}$. Only then, in the construction of the homotopy will a random number generator determine a constant γ. If the γ is predetermined, then it is possible to construct an input system $f(\mathbf{x}) = \mathbf{0}$ and a start system $g(\mathbf{x}) = \mathbf{0}$ for which there are bad values for $t \in [0, 1($. But reverting the usual order of operations is a bit similar to guessing the outcome of a coin toss after the coin toss and not before the coin toss. Therefore phc -0 should be used only for debugging purposes.

3.2.2 phc -a : solving polynomial systems equation-by-equation

The equation-by-equation solver applies the diagonal homotopies to intersect the solution set of the current set of polynomials with the next polynomial equation. The user has the possibility to shuffle the order of input polynomials.

Consider for example the following system:

```
3
 (x-1)*(x^2 - y)*(x-0.5);
 (x-1)*(x^3 - z)*(y-0.5);
 (x-1)*(x*y - z)*(z-0.5);
```

Because of its factored form, we see that its solution set contains

0. at least one isolated point $(0.5, 0.5, 0.5)$;
1. the twisted cubic $(y = x^2, z = x^3)$; and
2. the two dimensional plane defined by $x - 1 = 0$.

The output of `phc -a` will produce three files, with suffixes `_w0`, `_w1`, `_w2`, respectively for the zero dimensional, the one dimensional, and the two dimensional parts of the solution set.

0. a list of candidate isolated points;
1. generic points on the twisted cubic; and
2. one generic point on the plane $x = 1$.

The positive dimensional solution sets are each represented by a witness set. A *witness set* for a *k*-dimensional solution set of a system *f* consists of the system *f*, augmented with *k* linear equations with random coefficients, and solutions which satisfy the augmented system. Because the linear equations have random coefficients, each solution of the augmented system is a generic point. The number of generic points equals the degree of the solution set.

The output of `phc -a` gives a list of candidate witness points. In the example, the list of candidate isolated points will most likely contains points on higher dimensional solution sets. Such points can be filtered away with the homotopy membership test available in `phc -f`.

After filtering the points on higher dimensional solution sets, each pure dimensional solution set may decompose in irreducible components. The factorization methods of `phc -f` will partition the witness points of a pure dimensional solution set according to the irreducible factors.

The equation-by-equation solver gives *bottom up* way to compute a numerical irreducible decomposition. The diagonal homotopies can be called explicitly at each level with the option `-c`. The alternative *top down* way is available in `phc -c` as well.

3.2.3 phc -b : batch or blackbox processing

As a simple example of the input format for `phc -b`, consider the following three lines

```
2
 x**2 + 4*y**2 - 4;
        2*y**2 - x;
```

as the content of the file `input`. See the section on `phc -g` for a description of the input format.

To run the blackbox solver at the command line, type `phc -b input output`. The solutions of the system are appended to the polynomials in the file `input`. The file `output` also contains the solutions, in addition to more diagnostics about the solving, such as the root counts, start system, execution times.

The blackbox solver operates in four stages:

1. Preprocessing: scaling (`phc -s`), handle special cases such as binomial systems.
2. Counting the roots and constructing a start system (`phc -r`). Various root counts, based on the degrees and Newton polytopes, are computed. The blackbox solver selects the smallest upper bound on the number of isolated solution in the computation of a start system to solve the given polynomial system.
3. Track the solution paths from the solutions of the start system to the solutions of the target system (`phc -p`).
4. Verify whether all end points of the solution paths are distinct, apply Newton's method with deflation on singular solutions (`phc -v`).

Through the options `-s`, `-r`, `-p`, and `-v`, the user can go through the stages separately. See the documentation for `phc -v` for a description of the quality indicators for the numerically computed solutions.

The blackbox solver recognizes several special cases:

1. one polynomial in one variable;
2. one system of linear equations;
3. a system with exactly two monomials in every equation.

For these special cases, no polynomial continuation is needed.

Polyhedral homotopies can solve Laurent systems, systems where the exponents of the variables can be negative. If the system on input is a Laurent system, then polyhedral homotopies (see the documentation for `-m`) are applied directly and no upper bounds based on the degrees are computed.

New since version 2.4.02 are the options `-b2` and `-b4` to run the blackbox solver respectively in double double and quad double precision, for example as

```
phc -b2 cyclic7 /tmp/c7out2
phc -b4 cyclic7 /tmp/c7out4
```

The most computational intensive stage in the solver is in the path tracking. Shared memory multitasked path trackers are available in the path trackers for both the polyhedral homotopies to solve a random coefficient system and for the artificial-parameter homotopy towards the target system. See the documentation for the option `phc -t` below.

The focus on `-b` is on isolated solutions. For a numerical irreducible decomposition of all solutions, including the positive dimensional ones, consider the options `-a`, `-c`, and `-f`.

3.2.4 phc -B : numerical irreducible decomposition in blackbox mode

The `-B` option does currently the same as option `0` of `phc -c`, that is: track all path defined by a cascade homotopy to compute candidate generic points on all components of the solution set. In the near future, new releases will compute

a complete numerical irreducible decomposition.

3.2.5 phc -c : irreducible decomposition for solution components

In a numerical irreducible decomposition, positive dimensional solution sets are represented by a set of generic points that satisfy the given system and as many linear equations with random coefficients as the dimension of the solution set. The number of generic points in that so-called witness set then equals the degree of the solution set.

The menu structure for a numerical irreducible decomposition consists of three parts:

1. Running a cascade of homotopies to compute witness points.
2. Intersecting witness sets with diagonal homotopies.
3. For binomial systems, the irreducible decomposition yields lists of monomial maps.

For the cascade of homotopies, the first choice in the menu combines the next two ones. The user is prompted to enter the top dimension (which by default is the ambient dimension minus one) and then as many linear equations with random coefficients are added to the input system. In addition, as many slack variables are added as the top dimension. Each stage in the cascade removes one linear equation and solutions with nonzero slack variables at the start of the homotopy may end at solutions of lower dimension.

The classification of the witness points along irreducible factors may happen with the third menu choice or, using different methods, with `phc -f`. The third menu choice of `phc -c` applies bivariate interpolation methods, while `phc -f` offers monodromy breakup and a combinatorial factorization procedure.

The intersection of witness sets with diagonal homotopies may be performed with extrinsic coordinates, which doubles the total number of variables, or in an intrinsic fashion. The intersection of witness sets is wrapped in `phc -w`.

The third block of menu options of `phc -c` concerns binomial systems. Every polynomial equation in a binomial system has exactly two monomials with a nonzero coefficient. The positive dimensional solution sets of such a system can be represented by monomial maps. For sparse polynomial systems, monomial maps are much more efficient data structures than witness sets.

3.2.6 phc -d : linear and nonlinear reduction w.r.t. the total degree

Degree bounds for the number of isolated solution often overshoot the actual number of solution because of relationships between the coefficients. Consider for example the intersection of two circles. A simple linear reduction of the coefficient matrix gives an equivalent polynomial system (having the same number of affine solutions) but with lower degrees. Reducing polynomials to introduce more sparsity may also benefit polyhedral methods.

As an example, consider the intersection of two circles:

```
2
 x^2 + y^2 - 1;
 (x -  0.5)^2 + y^2 - 1;
```

A simple linear combination of the two polynomials gives:

```
2
 x^2 + y^2 - 1;
 x -  2.5E-1;
```

This reduced system has the same solutions, but only two instead of four solution paths need to be tracked.

Nonlinear reduction attempts to replace higher degree polynomials in the system by S-polynomials.

3.2.7 phc -e : SAGBI/Pieri/Littlewood-Richardson homotopies

Numerical Schubert calculus is the development of numerical homotopy algorithms to solve Schubert problems. A classical problem in Schubert calculus is the problem of four lines. Given four lines in three dimensional space, find all lines that meet the four given lines in a point. If the lines are in general position, then there are exactly two lines that satisfy the problem specification. Numerical homotopy continuation methods deform a given generic problem into special position, solve the problem in special position, and then deform the solutions from the special into the generic problem.

As Schubert calculus is a classical topic in algebraic geometry, what seems less well known is that Schubert calculus offers a solution to the output pole placement problem in linear systems control. The option `phc -k` offers one particular interface dedicated to the Pieri homotopies to solve the output pole placement problem. A related problem that can be solved with Schubert calculus is the completion of a matrix so that the completed matrix has a prescribed set of eigenvalues.

In numerical Schubert calculus, we have three types of homotopies:

1. SAGBI homotopies solve hypersurface intersection conditions the extrinsic way. The problem is: in n-space, where $n = m + p$, for mp given m-planes, compute all p-planes which meet the m-planes nontrivially.
2. Pieri homotopies are intrinsically geometric and are better able to solve more general problems in enumerate geometry. Pieri homotopies generalize SAGBI homotopies in two ways:
   (a) The intersection conditions may require that the planes meet in a space of a dimension higher than one. In addition to the m-planes, the intersection conditions contain the dimensions of the spaces of intersection.
   (b) The solutions may be curves that produce p-planes. The problem may then be formulated as an interpolation problem. Given are $mp+q(m+p)$ interpolation points and as many m-planes on input. The solutions are curves of degree q that meet the given m-planes at the given interpolation points.
3. Littlewood-Richardson homotopies solve general Schubert problems. On input is a sequence of square matrices. With each matrix corresponds a bracket of intersection conditions on p-planes. Each intersection condition is the dimension of the intersection of a solution p-plane with a linear space with generators in one of the matrices in the sequence on input.

The earliest instances of SAGBI and Pieri homotopies were already available in version 2.0 of PHCpack. Since version 2.3.95, a more complete implementation of the Littlewood-Richardson homotopies is available.

3.2.8 phc -f : factor a pure dimensional solution set into irreducibles

The `f` in `-f` is the `f` of factor and filter.

The first basic filter allows for example to extract the real solutions from a given list. Other filtering criteria involve for example the residual, the estimate of the inverse condition numbers, and a test whether a coordinate of a solution is zero or not.

The second filter implements the homotopy membership test to decide whether a point belongs to a witness set. This filter is needed to process the superwitness sets computed by `phc -a` or `phc -c`. Given on input a witness set and a point, this filter runs a homotopy to decide if the point belongs to the positive dimensional solution set represented by the given witness set.

The factorization method take on input a witness set for a pure dimensional solution set of a polynomial system. For small degrees, a combinatorial factorization method will be fast. The second factorization method applies monodromy loops, using the linear trace test as a stop criterion.

Another option in the menu of `phc -f` gives access to a tropical method to detect a common factor of two Laurent polynomials.

3.2.9 phc -g : check the format of an input polynomial system

The purpose of `phc -g` is to check whether a given input system has the right syntax. A related option is `phc -o`.

Use `-g` as `phc -g input output` where `input` and `output` are the names of input and output files respectively. If `output` is omitted, then the output can be written to screen. If both `input` and `output` are omitted, then the user will be prompted to provide the polynomials in the input system.

The input system can be a system of polynomials in several variables with complex coefficients. The first line on the input file must be the number of polynomials. If the number of variables is different from the number of polynomials, then the second number on the first line must be the number of variables. Variables may have negative exponents, in which case the system is recognized as a Laurent polynomial system. Working with negative exponents can be useful to exclude solutions with zero coordinates, as polyhedral homotopies (see `phc -m`) are capable of avoiding to compute those type of solutions.

The division operator `/` may not appear in a monomial, e.g.: `x/y` is invalid, but may be used in a coefficient, such as in `5/7`. While `phc -g` will parse `5/7` in double precision, `phc -v` will use the proper extended precision in its multiprecision root refinement.

The coefficients of the system will be parsed by `phc -g` as complex numbers in double precision. There is also no need to declare variables, the names of the variables will be added to the symbol table, in the order of which they occur in the polynomials in the system. A trick to impose an order on the variables is to start the first polynomial with the zero polynomial, written as `x - x + y - y`, to ensure that the symbol `x` comes prior to `y`. Internally, the terms in a polynomial are ordered in a graded lexicographical order.

Names that may not be used as names for variables are `e`, `E` (because of the scientific format of floating-point numbers) and `i`, `I` (because of the imaginary unit $\sqrt{-1}$). Every polynomial must be terminated by a semicolon, the `;` symbol. Starting the name of a variable with `;` is in general a bad idea anyway, but semicolons are used as terminating symbols in a polynomial.

Round brackets are for grouping the real and imaginary parts of complex coefficients, e.g.: `(1.e-3 + 3/7*I)*x^2*y` or for grouping factors, e.g.: `3.14*(x+y)*(x-1)^4`.

3.2.10 phc -h : writes helpful information to screen

The information written by `phc -h` is the condensed version of this document. For every option, some helpful information is defined. For example, typing `phc -z -h` or `phc -h -z` displays information about `phc -z`.

Typing `phc -h -h` displays the list of all available options.

Instead of `-h`, one can also type `--help`.

3.2.11 phc -j : path tracking with algorithmic differentiation

In the tracking of a solution path we frequently apply Newton's method. To run Newton's method we need to evaluate the system and compute all its partial derivatives. The cost of evaluation and differentiation is a significant factor in the total cost. For large systems, this cost may even dominate.

The `phc -j` gives access to the Path library developed to accelerate the path trackers with graphics processing units. The code is capable to evaluate and differentiate large polynomial systems efficiently, in double, double double, and quad double precision.

3.2.12 phc -k : realization of dynamic output feedback placing poles

The homotopies in numerical Schubert calculus (see the option -e) solve the output pole placement problem in linear systems control. The option `-k` applies the Pieri homotopies to compute feedback laws for plants defined by (A,B,C)

matrices.

For examples of input, see the `Feedback/Data` folder of the source code.

The feedback laws in the output file are realized and written in a format ready for parsing with MATLAB or Octave. The computation of output feedback laws is an application of the Pieri homotopies, available via `phc -e`.

3.2.13 phc -l : witness set for hypersurface cutting with random line

A hypersurface defined by a polynomial in several variables is cut with one general line. The number of points on the hypersurface and the general line equal the degree of the hypersurface. This collection of points on the intersection of a hypersurface and a general line form a witness set.

For example, if the file `sphere` contains

```
1 3
x^2 + y^2 + z^2 - 1;
```

then typing at the command prompt

```
phc -l sphere sphere.out
```

results in the creation of the file `sphere_w2` which contains a witness set of dimension two for the unit sphere. The output file `sphere.out` contains diagnostics about the computation.

For hypersurfaces of higher degree, the double precision as provided by the hardware may turn out to be insufficient to compute as many generic points as the degree of the hypersurface. Therefore, the options `l2` and `l4` perform the computations respectively in double double and quad double precision. To continue the example from above, typing at the command prompt

```
phc -l4 sphere sphere.qd
```

will give two generic points on the sphere, computed in quad double precision.

3.2.14 phc -m : mixed volume computation via lift+prune and MixedVol

The menu choices of `phc -m` are a subset of the menu of `phc -r`. The focus on `phc -m` is on mixed volumes. For polynomial systems with as many equations as unknowns, the mixed volume of the Newton polytopes gives a generically sharp upper bound on the number of isolated solutions with coordinates different from zero.

The ability to focus only on solutions with all coordinates different from zero stems from the fact that shifting Newton polytopes (which corresponds to multiplying the polynomials with one monomial) does not increase their volumes. With polyhedral homotopies, we can solve systems of polynomials with negative exponents for the variables, the so-called Laurent polynomials.

The mixed volume of a tuple of Newton polytopes is defined as the coefficient in the expansion of the volume of a linear combination of Newton polytopes. For example, for a 3-tuple of Newton polytopes:

$$\begin{aligned} vol(\lambda_1 P_1 + \lambda_2 P_2 + \lambda_3 P_3) &= V(P_1,P_1,P_1)\lambda_1^3 \\ &+ V(P_1,P_1,P_2)\lambda_1^2\lambda_2 \\ &+ V(P_1,P_2,P_2)\lambda_1\lambda_2^2 \\ &+ V(P_1,P_2,P_3)\lambda_1\lambda_2\lambda_3 \\ &+ V(P_2,P_2,P_2)\lambda_2^3 \\ &+ V(P_2,P_2,P_3)\lambda_2^2\lambda_3 \\ &+ V(P_2,P_3,P_3)\lambda_2\lambda_3^2 \\ &+ V(P_3,P_3,P_3)\lambda_3^3 \end{aligned}$$

where $vol(\cdot)$ is the volume function and $V(\cdot)$ is the mixed volume. For the tuple (P_1, P_2, P_3), its mixed volume is $V(P_1, P_2, P_3)$ in the expansion above.

The polynomial above can be called the Minkowski polynomial and with the Cayley trick we can compute all its coefficients. This is implemented with the dynamic lifting algorithm.

The menu with 5 different lifting strategies is displayed as follows:

```
MENU with available Lifting Strategies :
  0. Static lifting     : lift points and prune lower hull.
  1. Implicit lifting   : based on recursive formula.
  2. Dynamic lifting    : incrementally add the points.
  3. Symmetric lifting  : points in same orbit get same lifting.
  4. MixedVol Algorithm : a faster mixed volume computation.
Type 0, 1, 2, 3, or 4 to select, eventually preceded by i for info :
```

The menu of `phc -m` offers 5 different algorithms:

0. Static lifting: a lifting function is applied to the points in the support sets of the polynomials in the system and the lower hull defines the mixed cells. The users can specify the lifting values interactively. Liftings that do not lead to cells that are fine mixed are subdivided with a random lifting.
1. Implicit lifting: based on a recursive formula used in Bernshtein's original proof that the mixed volumes bounds the number of isolated solutions with nonzero coordinates.
2. Dynamic lifting: points are added one after the other in an incremental construction of a mixed cell configuration. An implementation of the Cayley trick gives the Minkowski polynomial.
3. Symmetric lifting: many systems have Newton polytopes that are invariant to permutation symmetry. Even if the original system is not symmetric, the construction of the start system could benefit from the exploitation of this permutation symmetry.
4. The MixedVol Algorithm is a specific implementation of the static lifting method, applying a floating random lifting function.

   The code offered with this option is a translation of software described in the paper by Tangan Gao, T. Y. Li, Mengnien Wu: *Algorithm 846: MixedVol: a software package for mixed-volume computation.* ACM Transactions on Mathematical Software, 31(4):555-560, 2005; distributed under the terms of the GNU General Public License as published by the Free Software Foundation.

   With the stable mixed volume we count *all* affine solutions (not only those with nonzero coordinates) and then and obtain polyhedral homotopies that compute all affine solutions.

On multicore computers, the solution of a random coefficient system with polyhedral homotopies runs in parallel when calling phc with the option `-t`. For example, `phc -m -t8` will run the polyhedral path trackers with 8 tasks. Since version 2.4.06, the mixed volume computation by the MixedVol algorithm (option 4 of `phc -m`) is interlaced with the path tracking in a heterogenous pipelined application of multitasking.

3.2.15 phc -o : writes the symbol table of an input system

Running `phc -o` with as input argument a polynomial system writes the symbols for the variables in the order in which they are stored internally after parsing the system. For example, if the file `/tmp/ex1` contains the lines

```
2
  y + x + 1;
  x*y - 1;
```

then running `phc -o` at the command prompt as

```
phc -o /tmp/ex1 /tmp/ex1.out
```

makes the file `/tmp/ex1.out` which contains the line

```
y x
```

because in the formulation of the polynomial system, the variable with name `y` occurred before the variable with name `x`. Consequently, the order of the coordinates of the solutions will then also be stored in the same order as of the occurrence of the variable names. If a particular order of variables would be inconvenient, then a trick to force an order on the variables is to insert a simple polynomial that simplifies to zero. For example, a modification of the file `/tmp/ex1` could be

```
2
 x + y - x - y +
 y + x + 1;
 x*y - 1;
```

and the first four monomials `x + y - x - y` will initialize the symbol table with the names x and y, in that order.

3.2.16 phc -p : polynomial continuation in one parameter

We distinguish between two types of homotopies. In an artificial parameter homotopy, the user is prompted for a target system and a start system with start solutions. If the input to `phc -p` is a polynomial system with one more unknown than the number of equations, then we have a natural parameter homotopy and the user is then prompted to define one unknown as the continuation parameter.

We first illustrate artificial parameter homotopy continuation. In the example below, the artificial parameter is denoted by t and, as t goes from zero to one, a simpler polynomial system, the start system, is deformed to the target system, the system we want to solve:

$$\gamma(1-t)\left(\left\{\begin{array}{c} x^2 - c_1 = 0 \\ y - c_2 = 0 \end{array}\right\}\right) + t\left(\left\{\begin{array}{c} x^2 + y^2 - 1 = 0 \\ x + y - 2 = 0 \end{array}\right\}\right),$$

where γ, c_1, and c_2 are constants, generated at random on the unit circle in the complex plane.

For this example, the file with the target system contains

```
2
 x^2 + y^2 - 1;
 x + y - 2;
```

and the start system is then stored in the file with contents

```
2
 x^2 +(-7.43124688174374E-01 - 6.69152970422862E-01*i);
 y +(-7.98423708079157E-01 + 6.02095990999051E-01*i);

THE SOLUTIONS :
2 2
=============================================================================
solution 1 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -9.33575033988799E-01  -3.58381997194074E-01
 y :  7.98423708079157E-01  -6.02095990999051E-01
== err :  0.000E+00 = rco :  1.000E+00 = res :  0.000E+00 ==
```

```
solution 2 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.33575033988799E-01   3.58381997194074E-01
 y :  7.98423708079157E-01  -6.02095990999051E-01
== err :  0.000E+00 = rco :  1.000E+00 = res :  0.000E+00 ==
```

The start system can be constructed with `phc -r`.

With `phc -p`, the user has full access to all numerical tolerances that define how close the numerical approximations have to stay along a solution path. By default, path tracking happens in double precision, but the user can increase the precision via the menu of the homotopy settings. At the command line, launching `phc` with the options `-p2` and `-p4` will run the path tracking respectively in double double and quad double precision.

To rerun a selection of solution paths, the user should submit a start system which contains only the start solutions of those paths that need to be recomputed. In a rerun, one must choose the same γ as in the previous run.

In addition to the artificial parameter increment-and-fix continuation, there is support for complex parameter continuation and real pseudo arc length path tracking with detection of singularities using the determinant of the Jacobian along the solution path.

To run pseudo arc length continuation, the user has to submit a system that has fewer equations than variables. For example, for a *real* sweep of the unit circle, the input would be

```
2 3
 x^2 + y^2 - 1;
 y*(1-s) + (y-2)*s;
```

where the last equation moves the line $y = 0$ to $y = 2$. The sweep will stop at the first singularity it encounters on the solution path, which in this case is the quadratic turning point at $(0, 1)$.

The corresponding list of solutions should then contain the following:

```
2 3
===========================================================================
solution 1 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -1.00000000000000E+00   0.00000000000000E+00
 y :  0.00000000000000E+00   0.00000000000000E+00
 s :  0.00000000000000E+00   0.00000000000000E+00
== err :  0.000E+00 = rco :  1.863E-01 = res :  0.000E+00 ==
solution 2 :
t :  0.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  1.00000000000000E+00   0.00000000000000E+00
 y :  0.00000000000000E+00   0.00000000000000E+00
 s :  0.00000000000000E+00   0.00000000000000E+00
== err :  0.000E+00 = rco :  1.863E-01 = res :  0.000E+00 ==
```

After launching the program as `phc -p` the user can determine the working precision. This happens differently for the two types of homotopies, depending on whether the parameter is natural or artificial:

1. For a natural parameter homotopy like the sweep, the user will be prompted explicitly to choose between double, double double, or quad double precision.

2. For an artificial parameter homotopy, the user can determine the working precision at the construction of the homotopy.

In both types of homotopies, natural parameter and aritificial parameter, the user can preset the working precision respectively to double double or quad double, calling the program as `phc -p2` or as `phc -p4`.

Since version 2.4.13, `phc -p` provides path tracking for overdetermined homotopies, where both target and start system are given as overconstrained systems and every convex linear combination between target and start system admits solutions.

3.2.17 phc -q : tracking solution paths with incremental read/write

For huge polynomial systems, all solutions may not fit in memory. The jumpstarting method for a polynomial homotopy does not require the computation of all solutions of the start system and neither does it keep the complete solution list in memory.

The `phc -q` is a byproduct of the distributed memory parallel path trackers with developed with the Message Passing Interface (MPI). Even if one is not concerned about memory use, `phc -q` is an example of program inversion. Instead of first completely solving the start system before tracking solution paths to the target system, one can ask for the next start solution whenever one wants to compute another solution of the target system.

The menu of types of supported homotopies is

```
MENU for type of start system or homotopy :
  1. start system is based on total degree;
  2. a linear-product start system will be given;
  3. start system and start solutions are provided;
  4. polyhedral continuation on a generic system;
  5. diagonal homotopy to intersect algebraic sets;
  6. descend one level down in a cascade of homotopies;
  7. remove last slack variable in a witness set.
```

The first four options concern isolated solutions of polynomial systems. To construct a start system based on total degree or a linear-product start system, use `phc -r`. The polyhedral continuation needs a mixed cell configuration, which can be computed with `phc -m`.

Options 5 and 6 deal with positive dimensional solution sets, see `phc -c`.

3.2.18 phc -r : root counting and construction of start systems

The root count determines the number of solution paths that are tracked in a homotopy connecting the input system with the start system that has as many solutions as the root count. We have an optimal homotopy to solve a given system if the number of solution paths equals the number of solutions of the system.

Methods to bound the number of isolated solutions of a polynomial system fall in two classes:

1. Bounds based on the highest degrees of polynomials and variable groupings.
2. Bounds based on the Newton polytopes of the polynomials in the system. See the documentation for `phc -m`.

The complete menu (called with cyclic 5-roots, with total degree 120) is shown below:

```
MENU with ROOT COUNTS and Methods to Construct START SYSTEMS :
  0. exit - current root count is based on total degree : 120
PRODUCT HOMOTOPIES based on DEGREES ------------------------------
  1. multi-homogeneous Bezout number               (one partition)
  2. partitioned linear-product Bezout number    (many partitions)
  3. general linear-product Bezout number          (set structure)
```

```
  4. symmetric general linear-product Bezout number (group action)
POLYHEDRAL HOMOTOPIES based on NEWTON POLYTOPES ------------------
  5. combination between Bezout and BKK Bound   (implicit lifting)
  6. mixed-volume computation                     (static lifting)
  7. incremental mixed-volume computation        (dynamic lifting)
  8. symmetric mixed-volume computation        (symmetric lifting)
  9. using MixedVol Algorithm to compute the mixed volume fast (!)
```

At the start, the current root count is the total degree, which is the product of the degrees of the polynomials in the system. The options 5 to 9 of the menu are also available in `phc -m`.

Three different generalizations of the total degree are available:

1. For a multi-homogeneous Bezout number, we split the set of variables into a partition. A classical example is the eigenvalue problem. When viewed as a polynomial system $\lambda x - Ax = 0$ we see quadratic equations. Separating the variable for the eigenvalue λ from the coordinates x of the eigenvectors turns the system into a multilinear problem and provides the correct root count.
2. In a partitioned linear-product Bezout number, we allow that the different partitions of the sets of variables are used for different polynomials in the system. This may lead to a lower upper bound than the multi-homogeneous Bezout number.
3. A general linear-product Bezout number groups the variables in a collection of sets where each variable occurs at most once in each set. Every set then corresponds to one linear equation.

Each of these three generalizations leads to a linear-product start system. Every start solution is the solution of a linear system. One can view the construction of a linear-product start system as the degeneration of the given polynomial system on input such that every input polynomial is degenerated to a product of linear factors.

The fourth option of the `-r` allows to take permutation symmetry into account to construct symmetric start systems. If the start system respects the same permutation symmetry as the system on input, then one must track only those paths starting at the generators of the set of start solutions.

After the selection of the type of start system, the user has the option to delay the calculation of all start solutions. All start solutions can be computed at the time when needed by `phc -q`. To use the start system with `phc -p`, the user must ask to compute all start solutions with `-r`.

3.2.19 phc -s : equation and variable scaling on system and solutions

A system is badly scaled if the difference in magnitude between the coefficients is large. In a badly scaled system we observe very small and very large coefficients, often in the same polynomial. The solutions in a badly scaled system are ill conditioned: small changes in the input coefficients may lead to huge changes in the coordinates of the solutions.

Scaling is a form of preconditioning. Before we solve the system, we attempt to reformulate the original problem into a better scaled one. We distinguish two types of scaling:

1. equation scaling: multiply every coefficient in the same equation by the same constant; and
2. variable scaling: multiply variables by constants.

Chapter 5 of the book of Alexander Morgan on *Solving Polynomial Systems Using Continuation for Engineering and Scientific Problems* (volume 57 in the SIAM Classics in Applied Mathematics, 2009) describes the setup of an optimization problem to compute coordinate transformations that lead to better values of the coefficients.

If the file `/tmp/example` contains the following lines

```
2
 0.000001*x^2 + 0.000004*y^2 - 4;
 0.000002*y^2 - 0.001*x;
```

then a session with `phc -s` (at the command prompt) to scale the system goes as follows.

```
$ phc -s
Welcome to PHC (Polynomial Homotopy Continuation) v2.3.99 31 Jul 2015
Equation/variable Scaling on polynomial system and solution list.

MENU for the precision of the scalers :
  0. standard double precision;
  1. double double precision;
  2. quad double precision.
Type 0, 1, or 2 to select the precision : 0

Is the system on a file ? (y/n/i=info) y

Reading the name of the input file.
Give a string of characters : /tmp/example

Reading the name of the output file.
Give a string of characters : /tmp/example.out

MENU for Scaling Polynomial Systems :
  1 : Equation Scaling : divide by average coefficient
  2 : Variable Scaling : change of variables, as z = (2^c)*x
  3 : Solution Scaling : back to original coordinates
Type 1, 2, or 3 to select scaling, or i for info : 2
  Reducing the variability of coefficients ? (y/n) y
  The inverse condition is  4.029E-02.

Do you want the scaled system on separate file ? (y/n) y
Reading the name of the output file.
Give a string of characters : /tmp/scaled

$
```

Then the contents of the file `/tmp/scaled` is

```
2
x^2+ 9.99999999999998E-01*y^2-1.00000000000000E+00;
y^2-1.00000000000000E+00*x;

SCALING COEFFICIENTS :

10
3.30102999566398E+00   0.00000000000000E+00
3.00000000000000E+00   0.00000000000000E+00
-6.02059991327962E-01   0.00000000000000E+00
-3.01029995663981E-01   0.00000000000000E+00
```

We see that the coefficients of the scaled system are much nicer than the coefficients of the original problem. The scaling coefficients are needed to transform the solutions of the scaled system into the coordinates of the original problem. To transform the solutions, choose the third option of the second menu of `phc -s`.

3.2.20 phc -t : tasking for tracking paths using multiple threads

The problem of tracking a number of solution paths can be viewed as a pleasingly parallel problem, because the paths can be tracked independently from each other.

The option `-t` allows the user to take advantage of multicore processors. For example, typing at the command prompt.

```
phc -b -t4 cyclic7 /tmp/cyclic7.out
```

makes that the blackbox solver uses 4 threads to solve the system. If there are at least 4 computational cores available, then the solver may finish its computations up to 4 times faster than a sequential run.

With the time command, we can compare the wall clock time between a sequential run and a run with 16 tasks:

```
time phc -b cyclic7 /tmp/cyc7t1

real    0m10.256s
user    0m10.202s
sys     0m0.009s

time phc -b -t16 cyclic7 /tmp/cyc7t16

real    0m0.851s
user    0m11.149s
sys     0m0.009s
```

The speedup on the wall clock time is about 12, obtained as 10.256/0.851.

The relationship with double double and quad double precision is interesting, consider the following sequence of runs:

```
time phc -b cyclic7 /tmp/c7out1

real    0m9.337s
user    0m9.292s
sys     0m0.014s

time phc -b -t16 cyclic7 /tmp/c7out2

real    0m0.923s
user    0m13.034s
sys     0m0.010s
```

With 16 tasks we get about a tenfold speedup, but what if we ask to double the precision?

```
time phc -b2 -t16 cyclic7 /tmp/c7out3

real    0m4.107s
user    0m59.164s
sys     0m0.018s
```

We see that with 16 tasks in double precision, the elapsed time equals 4.107 seconds, whereas the time without tasking was 9.337 seconds. This means that with 16 tasks, for this example, we can double the working precision and still finish the computation is less than half of the time without tasking. We call this quality up.

For quad double precision, more than 16 tasks are needed to offset the overhead caused by the quad double arithmetic:

```
time phc -b4 -t16 cyclic7 /tmp/c7out4

real    0m53.865s
user    11m56.630s
sys     0m0.248s
```

To track solution paths in parallel with `phc -p`, for example with 4 threads, one needs to add `-t4` to the command line and call `phc` as `phc -p -t4`. The option `-t` can also be added to `phc -m` at the command line, to solve random coefficient start systems with polyhedral homotopies with multiple tasks.

3.2.21 phc -u : Newton's method for power series solution

The application of Newton's method over the field of truncated power series in double, double double, or quad double precision, can be done with `phc -u`.

On input is a polynomial system where one of the variables will be considered as a parameter in the series. The other input to `phc -u` is a list of solution for the zero value of the series variable.

Consider for example the intersection of the Viviani curve with a plane, as defined in the homotopy:

```
3 4
 (1-s)*y + s*(y-1);
 x^2 + y^2 + z^2 - 4;
 (x-1)^2 + y^2 - 1;
```

At `s=0`, the point `(0,0,2)` is a regular solution and the file with the homotopy should contain

```
solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 s :  0.00000000000000E+00   0.00000000000000E+00
 y :  0.00000000000000E+00   0.00000000000000E+00
 x :  0.00000000000000E+00   0.00000000000000E+00
 z :  2.00000000000000E+00   0.00000000000000E+00
== err :  0.000E+00 = rco :  3.186E-01 = res :  0.000E+00 ==
```

The input file can be prepared inserting the `s=0` into the homotopy and giving to the blackbox solver `phc -b` a file with contents:

```
4
 s;
 (1-s)*y + s*(y-1);
 x^2 + y^2 + z^2 - 4;
 (x-1)^2 + y^2 - 1;
```

The output of `phc -b` will have the point `(0,0,2)` for `s=0`.

3.2.22 phc -v : verification, refinement and purification of solutions

While solution paths do in general not become singular or diverge, at the end of the paths, solutions may turn out to be singular and/or at infinity.

Consider for example the system

```
2
 x*y + x - 0.333;
 x^2 + y - 1000;
```

where the first solution obtained by some run with `phc -b` is

```
solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  3.32667332704111E-04  -2.78531008415435E-26
 y :  9.99999999889332E+02   0.00000000000000E+00
== err :  3.374E-09 = rco :  2.326E-06 = res :  3.613E-16 ==
```

The last three numbers labeled with `err`, `rco`, and `res` are indicators for the quality of the solution:

1. `err` : the magnitude of the last correction made by Newton's method to the approximate solution. The `err` measures the *forward error* on the solution. The forward error is the magnitude of the correction we have to make to the approximate solution to obtain the exact solution. As the value of `err` is about `1.0e-9` we can hope to have about eight correct decimal places in the solution.
2. `rco` : an estimate for the inverse of the *condition number* of the Jacobian matrix at the approximation for the solution. A condition number measures by how much a solution may change as by a change on the input coefficients. In the example above, the `0.333` could be a three digit approximation for `1/3`, so the error on the input could be as large as `1.0e-4`. As `rco` is about `1.0e-6`, the condition number is estimated to be of order `1.0e+6`. For this example, an error of 10^{-4} on the input coefficients can result in an error of $10^{-4} \times 10^{6} = 10^{2}$ on the solutions.
3. `res` : the magnitude of the polynomials in the system evaluated at the approximate solution, the so-called *residual*. For this problem, the residual is the *backward error*. Because of numerical representation errors, we have not solved an exact problem, but a nearby problem. The backward error measures by much we should change the input coefficients for the approximate solution to be an exact solution of a nearby problem.

With `phc -v` one can do the following tasks:

1. Perform a basic verification of the solutions based on Newton's method and weed out spurious solutions. The main result of a basic verification is the tally of good solutions, versus solutions at infinity and/or singular solutions. Solution paths may also have ended at points that are clustered at a regular solution so with '-v' we can detect some cases of occurrences of path crossing.

   To select solutions subject to given criteria, use `phc -f`.
2. Apply Newton's method with multiprecision arithmetic. Note that this may require that also the input coefficients are evaluated at a higher precision.
3. For isolated singular solutions, the deflation method may recondition the solutions and restore quadratic convergence. Note that a large condition number may also be due to a bad scaling of the input coefficients.

   With `phc -s` one may improve the condition numbers of the solutions.
4. Based on condition number estimates the working precision is set to meet the wanted number of accurate decimal places in the solutions when applying Newton's method.

The blackbox version uses default settings for the parameters, use as `phc -v -b` or `phc -b -v`, for double precision. For double double precision, use as `phc -b2 -v` or `phc -b -v2`. For quad double precision, use as `phc -b4 -v` or `phc -b -v4`. The order of `-b` and `-v` at the command line does not matter.

3.2.23 phc -w : witness set intersection using diagonal homotopies

This option wraps the diagonal homotopies to intersect two witness sets, see the option `-c` for more choices in the algorithms.

For example, to intersect the unit sphere (see the making of `sphere_w2` with `phc -l`) with a cylinder to form a quartic curve, we first make a witness set for a cylinder, putting in the file `cylinder` the two lines:

```
1 3
x^2 + y - y + (z - 0.5)^2 - 1;
```

Please note the introduction of the symbol `y` even though the symbol does not appear in the equation of a cylinder about the y-axis. But to intersect this cylinder with the unit sphere the symbols of both witness sets must match. After executing `phc -l cylinder cylinder.out` we get the witness set `cylinder_w2` and then we intersect with `phc -w`:

```
phc -w sphere_w2 cylinder_w2 quartic
```

The file `quartic` contains diagnostics of the computation. Four general points on the quartic solution curve of the intersection of the sphere and the cylinder are in the file `quartic_w1` which represents a witness set.

3.2.24 phc -x : convert solutions from PHCpack into Python dictionary

To work with solution lists in Python scripts, running `phc -x` converts a solution list in PHCpack format to a list of dictionaries. Given a Python list of dictionaries, `phc -x` returns a list of solutions in PHCpack format. For example:

```
phc -x cyclic5 /tmp/cyclic5.dic
phc -x /tmp/cyclic5.dic
```

The first `phc -x` writes to the file `/tmp/cyclic5.dic` a list of dictionaries, ready for processing by a Python script. If no output file is given as second argument, then the output is written to screen. The second `phc -x` writes a solution list to PHCpack format, because a list of dictionaries is given on input.

If the second argument of `phc -x` is omitted, then the output is written to screen. For example, if the file `/tmp/example` contains

```
2
 x*y + x - 3;
 x^2 + y - 1;

THE SOLUTIONS :
3 2
===========================================================================
solution 1 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x : -1.89328919630450E+00   0.00000000000000E+00
 y : -2.58454398084333E+00   0.00000000000000E+00
== err :  2.024E-16 = rco :  2.402E-01 = res :  2.220E-16 ==
solution 2 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.46644598152249E-01  -8.29703552862405E-01
 y :  7.92271990421665E-01   1.57086877276985E+00
== err :  1.362E-16 = rco :  1.693E-01 = res :  2.220E-16 ==
solution 3 :
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
the solution for t :
 x :  9.46644598152249E-01   8.29703552862405E-01
 y :  7.92271990421665E-01  -1.57086877276985E+00
== err :  1.362E-16 = rco :  1.693E-01 = res :  2.220E-16 ==
```

then the conversion executed by

```
phc -x /tmp/example
```

write to screen the following:

```
[
{'time': 1.00000000000000E+00 +  0.00000000000000E+00*1j, \
'multiplicity':1,'x':-1.89328919630450E+00 +  0.00000000000000E+00*1j, \
'y':-2.58454398084333E+00 +  0.00000000000000E+00*1j, \
'err':  2.024E-16,'rco':  2.402E-01,'res':  2.220E-16}, \
{'time': 1.00000000000000E+00 +  0.00000000000000E+00*1j, \
'multiplicity':1,'x': 9.46644598152249E-01-8.29703552862405E-01*1j, \
'y': 7.92271990421665E-01 +  1.57086877276985E+00*1j, \
'err':  1.362E-16,'rco':  1.693E-01,'res':  2.220E-16}, \
{'time': 1.00000000000000E+00 +  0.00000000000000E+00*1j, \
'multiplicity':1,'x': 9.46644598152249E-01 +  8.29703552862405E-01*1j, \
'y': 7.92271990421665E-01-1.57086877276985E+00*1j, \
'err':  1.362E-16,'rco':  1.693E-01,'res':  2.220E-16}
]
```

In the output above, for readabiility, extra line breaks were added, after each continuation symbol (the back slash). In the output of `phc -x`, every dictionary is written on one single line.

The keys in the dictionary are the same as the left hand sides in the equations in the Maple format, see `phc -z`.

3.2.25 phc -y : sample points from an algebraic set, given witness set

The points on a positive dimensional solution set are fixed by the position of hyperplanes that define a linear space of the dimension equal to the co-dimension of the solution set. For example, in 3-space, a 2-dimensional set is cut with a line and a 1-dimensional set is cut with a plane.

Given in `sphere_w2` a witness set for the unit sphere (made with `phc -l`, see above), we can make a new witness set with `phc -y`, typing at the command prompt:

```
phc -y sphere_w2 new_sphere
```

and answering two questions with parameter settings (type 0 for the defaults). The output file `new_sphere` contains diagnostics of the run and a new witness set is in the file `new_sphere_w2`.

3.2.26 phc -z : strip phc output solution lists into Maple format

Parsing solution lists in PHCpack format can be a bit tedious. Therefore, the `phc -z` defines a simpler format, representing a list of solutions as a list of lists, where lists are enclosed by square brackets. Every solution is a list of equations, using a comma to separate the items in the list.

The phc -z commands converts solution lists in PHCpack format into Maple lists and converts Maple lists into solutions lists in PHCpack format. For example:

```
phc -z cyclic5 /tmp/cyclic5.mpl
phc -z /tmp/cyclic5.mpl
```

If the file `cyclic5` contains the solutions of the cyclic 5-roots problem in PHCpack format, then the first command makes the file `/tmp/cyclic5.mpl` which can be parsed by Maple. The next command has no second argument for output file and the output is written directly to screen, converting the solutions in Maple format into solution lists in PHCpack format.

If the output file is omitted, then the output is written to screen. For example, if the file `/tmp/example` has as content

```
2
 x*y + x - 3;
 x^2 + y - 1;
```

Then we first can solve the system with the blackbox solver as

```
phc -b /tmp/example /tmp/example.out
```

Because `phc -b` appends the solution to an input file without solutions, we can convert the format of the PHCpack solutions into Maple format as follows:

```
phc -z /tmp/example
[[time = 1.0 + 0*I,
  multiplicity = 1,
  x = -1.8932891963045 + 0*I,
  y = -2.58454398084333 + 0*I,
  err =  2.024E-16,  rco =  2.402E-01,  res =  2.220E-16],
 [time = 1.0 + 0*I,
  multiplicity = 1,
  x = 9.46644598152249E-1 - 8.29703552862405E-1*I,
  y = 7.92271990421665E-1 + 1.57086877276985*I,
  err =  1.362E-16,  rco =  1.693E-01,  res =  2.220E-16],
 [time = 1.0 + 0*I,
  multiplicity = 1,
  x = 9.46644598152249E-1 + 8.29703552862405E-1*I,
  y = 7.92271990421665E-1 - 1.57086877276985*I,
  err =  1.362E-16,  rco =  1.693E-01,  res =  2.220E-16]];
```

The left hand sides of the equations are the same as the keys in the dictionaries of the Python format, see `phc -x`.

CHAPTER 4

Reference Manual

The code is written in the following languages: Ada, C, C++ (NVIDIA CUDA), and Python. The following description documents the organization and design decisions which led to the current state of the code.

The main executable `phc` compiles on Linux, MacOS X, and Windows computers. Shared memory parallelism works on all three operating systems. The message passing with MPI has not been tested on Windows. The development of the accelerator code with NVIDIA CUDA was done on Linux computers.

The Python code was developed and tested on Linux and MacOS X, not on Windows.

4.1 The Test Procedures

A dynamic manner to experience the structure of the source code is to run through all test procecures. There over three hundred test programs which can be built typing

```
make testall
```

at the command prompt when situated in the `Objects` directory. The tests are organized along the source code directories. Every directory in the source code hierarchy has its own test procedures which focus on the particular functionality coded in that directory. To test the mathematical library, running `make test_math_lib` invokes ten other makes, one for each subdirectory of the mathematical library.

4.2 Organization of the Ada code

The code in the first release was written in Ada 83. Release 2 featured a new mathematical library, rebuilt using Ada 95 concepts and offering multi-precision arithmetic.

There are four major layers in the code:

1. `Math_Lib`: linear algebra, representations of polynomials, Newton polytopes, and power series;
2. `Deformations`: Newton's method, path trackers, end games, solutions and homotopies, deflation;

3. `Root_Counts`: root counting methods and constructions of homotopies, linear-product start start systems based on Bezout bounds, mixed volumes and polyhedral homotopies;
4. `Components`: witness sets, cascades of homotopies, monodromy, diagonal homotopies, to compute a numerical irreducible decomposition.

There are five other parts, called `System`, `Schubert`, `CtoPHC`, `PHCtoC`, and `Tasking`. The top down perspective starts at the folder `Main`.

The Ada sources are organized in a tree of directories:

```
Ada                       : Ada source code of PHC
 |-- System                : 0. OS dependencies, e.g.: timing package
 |-- Math_Lib              : 1. general mathematical library
 |       |-- Numbers       : 1.1. number representations
 |       |-- QD            : 1.2. quad doubles
 |       |-- Vectors       : 1.3. vectors and vectors of vectors
 |       |-- Matrices      : 1.4. matrices and linear-system solvers
 |       |-- Divisors      : 1.5. common divisors, integer linear algebra
 |       |-- Reduction     : 1.6. row reduction, numerical linear algebra
 |       |-- Polynomials   : 1.7. multivariate polynomial systems
 |       |-- Functions     : 1.8. evaluation and differentiation
 |       |-- Supports      : 1.9. support sets and linear programming
 |       |-- Circuits      : 1.A. circuits for algorithmic differentation
 |       |-- Series        : 1.B. manipulating truncated series
 |-- Deformations          : 2. homotopies, Newton's method & path trackers
 |       |-- Solutions     : 2.1. solutions of systems and homotopies
 |       |-- Homotopy      : 2.2. homotopies, scaling and reduction
 |       |-- Newton        : 2.3. root refining and modified Newton's method
 |       |-- Curves        : 2.4. univariate solving & plane algebraic curves
 |       |-- End_Games     : 2.5. extrapolation end games with Puiseux series
 |       |-- Trackers      : 2.6. path-tracking routines
 |       |-- Sweep         : 2.7. sweeping for singularities
 |       |-- Continuation  : 2.8. drivers and data management
 |-- Root_Counts           : 3. root counts and homotopy construction
 |       |-- Product       : 3.1. linear-product start systems
 |       |-- Binomials     : 3.2. solvers for binomial and simplicial systems
 |       |-- Implift       : 3.3. implicit lifting
 |       |-- Stalift       : 3.4. static lifting
 |       |-- Dynlift       : 3.5. dynamic lifting
 |       |-- Symmetry      : 3.6. exploitation of symmetry relations
 |       |-- MixedVol      : 3.7. translation of ACM TOMS Algorithm 846
 |       |-- Puiseux       : 3.8. Puiseux series for curves
 |-- Schubert              : 4. numerical Schubert calculus
 |       |-- SAGBI         : 4.1. SAGBI homotopies
 |       |-- Pieri         : 4.2. deformations based on Pieri's rule
 |       |-- Induction     : 4.3. Schubert induction
 |-- Components            : 5. numerical irreducible decomposition
 |       |-- Samplers      : 5.1. computing witness points
 |       |-- Interpolators : 5.2. finding equations for components
 |       |-- Factorization : 5.3. factorization into irreducible components
 |       |-- Decomposition : 5.4. sequence of homotopies to filter and factor
 |       |-- Solver        : 5.5. incremental equation by equation solver
 |       |-- Tropical      : 5.6. tropical view on witness sets
 |-- CtoPHC                : 6. interface from C to phc
 |       |-- Funky         : 6.1. functional interface, C -> Ada -> C
 |       |-- State         : 6.2. state machine gateway, C <-> Ada
 |-- PHCtoC                : 7. GPU acceleration via a C interface
 |-- Tasking               : 8. multitasking
```

```
|-- Main                    : 9. main dispatcher
```

Every directory contains a collection of test procedures. The following sections describe the functionality defined in each of the directories. Then the subdirectories are described in separate sections.

4.2.1 System: OS Dependencies such as Timing

The `System` directory defines operations that may have different definitions on different operation systems. One such operation is to compute the elapsed CPU time of a computation. The timer for Ada on Unix like operation systems was originally developed by Dave Emory of the MITRE corporation. Not everything in this timing package could be mapped to Windows, in particular the resource usage report for Unix. While the interface of the timing package is the same for all operating systems, the implementation differs for Windows

When multithreaded runs on multicore processors, the elapsed CPU time is most often not a good time measurement and one comes interested in the wall clock time. The end of the output contains the start and end date of the computation. With the `Ada.Calendar`, the time stamping is defined in a portable, operating system independent manner.

The directory system contains several very useful utilities, such as procedures to prompt the user for a yes or no answer, or for a selection between various alternatives. While restricting the user selection, the prompting procedures allow to retry in case of type errors. Similar user friendly guards are defined when the user gives the name of an existing file for output. Before overwriting the existing file, the user is prompted to confirm. When reading a file, the user is allowed to retry in case the given name of the file does not match an existing file.

The handling of the command line options is also defined in this directory. Thanks to the `Ada.Command_Line`, this definition is operating system independent.

The package `machines` wraps some system calls. One such system call is to get the process identification number (pid). This pid is used to seed the random number generators.

4.2.2 The Mathematical Library

The mathematical library defines code that is not specific to polynomial homotopy continuation, but nevertheless necessary. To make PHCpack self contained, the code does not require the installation of outside libraries. Although there are eleven subdirectories, there are three main parts:

1. number representations, general multiprecision and quad doubles;
2. linear algebra with integers and floating-point numbers;
3. polynomials, polynomial functions, series, and Newton polytopes.

The input to a polynomial system solver is a list of polynomials in several variables. This input consists of exact data, such as the integer exponents in the monomials, and approximate data, such as the floating-point coefficients of the monomials. Solving a polynomial system with homotopy continuation is therefore always a hybrid computation, involving exact and approximate data. While the machine arithmetic may still suffice for many applications, the increasing available computational power has led to the formulation of large problems for which software defined multiprecision arithmetic is required. The linear algebra operations are defined over exact number rings and over arbitrary precision floating-point numbers.

The next subsections contain more detailed descriptions of each subdirectory of the mathematical library. The following three paragraphs briefly summarize the eleven subdirectories in the three main parts.

The number representations are defined in the subdirectory `Numbers` and the QD library of Y. Hida, X. S. Li, and D. H. Bailey is integrated in the subdirectory `QD`.

The linear algebra data structures are defined in the subdirectories `Vectors` and `Matrices`. The `Divisors` subdirectory relies on the greatest common divisor algorithm to define the Hermite and Smith normal forms to solve linear systems over the integer numbers. The linear system solvers of numerical linear algebra are provided in the subdirectory `Reduction`.

The third main part of the mathematical library consists in the remaining five of the eleven subdirectories. Multivariate polynomials over various number rings in the subdirectory `Polynomials`. The subdirectory `Functions` contains definitions of nested Horner schemes to efficiently evaluate dense polynomials. The support of a polynomial is the set of exponents of the monomials which appear with nonzero coefficients. Basic linear programming and tools to work with polytopes are provided in the subdirectory `Supports`. The subdirectory `Circuits` defines arithmetic circuits to evaluate and differentiate polynomials via the reverse mode of algorithmic differentiation. Truncated power series define a field (that is: dividing two series gives again a series) and the arithmetic to manipulate power series is exported by the packages in the subdirectory `Series`.

4.2.3 Deforming Polynomial Systems

A homotopy is a family of polynomial systems defined by one parameter. The parameter may be introduced in an artificial manner, such as the parameter t in the classical homotopy

$$h(\mathbf{x}, t) = (1 - t)g(\mathbf{x}) + tf(\mathbf{x}) = \mathbf{0}.$$

The homotopy $h(\mathbf{x}, t)$ connects the system $g(\mathbf{x}) = \mathbf{0}$ (the so-called *start system*) to the system $f(\mathbf{x}) = \mathbf{0}$ (the so-called *target system*), as $h(\mathbf{x}, 0) = g(\mathbf{x})$ and $h(\mathbf{x}, 1) = f(\mathbf{x})$. The solutions $\mathbf{x}(t)$ to the homotopy are solution paths, starting at $t = 0$ at the solutions of the start system and ended at $t = 1$ at the solutions of the target system.

The code was developed mainly for constructing artificial-parameter homotopies, but there is some still limited support for polynomial homotopies with natural parameters. Artificial-parameter homotopies can be constructed so that singular solutions occur only at the end of the paths. For natural-parameter homotopies, the detection and accurate computation of singularities along the paths becomes an important topic.

There are eight subdirectories in the `Deformations` directory. The subdirectories `Solutions` and `Homotopies` provide the data structures for the solutions on the paths defined by the polynomial homotopies. Newton's method and deflation are implemented in the subdirectory `Newton`. In `Curves` are the extrapolation methods for the predictors in the path trackers. Extrapolation for winding numbers is coded in the subdirectory `End_Games`. Path trackers for artificial-parameter homotopies are available in the `Trackers` subdirectory. In `Sweep` arc length parameter continuation is implemented for sweeping solution paths for singularities. Finally, the subdirectory `Continuation` contains the data management and driver procedures.

Observe that in the layered organization of the source code, the `Deformations` directory is placed before the `Root_Counts` directory, where the start systems are defined. This organization implies that the path trackers are written independently from the constructors for the polynomial homotopies.

4.2.4 Homotopy Construction via Root Counting Methods

At first, it seems counter intuitive to construct a polynomial homotopy to solve an unknown system by counting its roots. But consider the degeneration of two planar quadrics into lines. Each quadric degenerates to a pair of lines. How many solutions could we get intersection two pairs of lines in general position? Indeed, four, computed as two by two. Observe that in this simple argument we have no information about the particular representation of the quadrics. To get to this root count, we assumed only that the lines after degeneration were generic enough and the count involved only the degrees of the polynomials.

Of critical importance for the performance of a polynomial homotopy is the accuracy of the root count. If the root count is a too large upper bound for the number of solutions of the system that will be solved, then too many solution paths will diverge to infinity, representing a very wasteful computation.

We can construct homotopies based on the degree information alone or rely on the Newton polytopes. Sparse polynomial systems are systems where relatively few monomials appear with nonzero coefficient, relative to the degrees of the polynomials in the system. For sparse system, the information of the Newton polytopes provides a much sharper root count than the ones provided by the degrees.

There are eight subdirecties in the `Root_Counts` directory. Total degree and linear-product start systems are constructed in the subdirectory `Product`. The subdirectory `Binomials` provides solvers for the sparsest polynomial systems. The subdirectories `Implift`, `Stalift`, and `Dynlift` implement polyhedral homotopies, respectively with implicit, static, and dynamic lifting methods. In `MixedVol` is an adaptation of a fast mixed volume calculator. Code to exploit permutation symmetry is in the subdirectory `Symmetry`. A generalization of the Newton-Puiseux algorithm is implemented in the subdirectory `Puiseux`.

4.2.5 Numerical Schubert Calculus

The classical problem in Schubert calculus asks for the number of lines which meet four given general lines in 3-space. With polynomial homotopies, we not only count, but also compute the actual number of solutions to a Schubert problem.

The problem of four lines is a special case of a Pieri problem: compute all *p*-planes which meet $m \times p$ given *m*-planes in a space of dimension $m + p$. If the given *m*-planes are sufficiently generic, then all solution *p*-planes are isolated and finite in number. Pieri homotopies solve the output pole placement problem in linear systems control.

There are three subdirectories to the `Schubert` directory, each exporting a different type of homotopy to solve Schubert problems. The subdirectory `SAGBI` applies the concept of subalgebra analog to Groebner basis for ideals with polyhedral homotopies to solve Pieri problems. Pieri homotopies are defined in the subdirectory `Pieri`. The subdirectory `Induction` implements a geometric Littlewood-Richardson rule to solve general Schubert problems.

4.2.6 Numerical Irreducible Decomposition

Two important characteristics of a pure dimensional solution set of a polynomial system are its dimension and its degree. The dimension of a solution set equals the number of general linear equations we need to add to the polynomial system so the intersection of the solution set of the system with the hyperplanes consists of isolated points. The degree of a solution set then equals the number of isolated points we find after intersecting the solution set with as many general hyperplanes as the dimension of the set. These two characteristics are encoded in the *witness set* representation of a pure dimensional solution set. Given a polynomial system, a numerical irreducible decomposition of its solution set provides a witness set for each irreducible components, over all dimensions.

The decomposition can be computed in a top down fashion, with cascades of homotopies, starting a the top dimension. The bottom up computation applies diagonal homotopies. Systems can be solved equation-by-equation or subsystem-by-subsystem.

Three types of factorization methods are implemented. Interpolation with multivariate polynomials of increasing degrees is a local procedure. The second method runs monodromy loops to connect generic points on the same irreducible component, using the linear trace test as stop criterion. Thirdly, we can apply the linear trace test combinatorially, which often works very well for components of modest degrees.

The are six subdirectories of the `Components` directory. The `Samplers` subdirectory contains the definitions of the data structures to store witness sets. The multivariate interpolation algorithms are implemented in the `Interpolators` subdirectory. The subdirectory `Factorization` provides monodromy factorization and the linear trace test. Cascades of homotopies and diagonal homotopies are implemented in the subdirectory `Decomposition`. The `Solver` subdirectory provides an equation-by-equation solver. Finally, the `Tropical` subdirectory offers code to generalize the polyhedral homotopies from isolated solutions to the computation of representations of positive dimensional solution sets.

4.2.7 Calling Ada Code From C

The directory CtoPHC has two subdirectories, Funky and State, which define two different types of interfacing the Ada code with C. The first type is a functional interface, the second type is an interface which operates as a state machine.

In a functional interface, the main C program calls an Ada function, which then calls a C function to process the results computed by the Ada function. This interface was developed for the application of the Pieri homotopies to compute output feedback laws for linear systems control. This type of interface is direct and efficient. Its main application is in the Feedback folder which defines C functions to compute realizations of the computed feedback laws.

The goal of the state interface in the subdirectory State is to export all functionality of the Ada code to the C (and C++) programmer. The subdirectory State contains the definition of the use_c2phc function, which defines more than 700 jobs. The implementation of this function relies on various container packages which hold the persistent objects, mainly polynomial systems and solution lists.

If the main program is not an Ada procedure, but a C function, then adainit and adafinal must be called by the C code, respectively at the beginning and at the end of the computations. The code for adainit is generated by the binder, by gnatbind, which is executed before the linking. If the linking happens with the linker of the gnu-ada compiler, the gnatlink (as is the default), then gnatlink compiles the output of gnatbind. Otherwise, if the linking is done by another C compiler, we must explicitly compile the output of the binder, so the object code for the adainit can be linked as well. These observations are important in building a shared object with statically compiled Ada code. The shared object can then be used on systems where the gnu-ada compiler is not installed. The makefile_unix in the Objects directory contains the precise compilation instructions for Linux systems.

4.2.8 Calling C Code From Ada

The directory PHCtoC was set up to call the GPU code via a C interface. In its current state it defines the wrappers to call the accelerated path trackers with algorithmic differentiation. Its main goal is to define the extension modules for calling the accelerated path trackers from the Python package phcpy.

As a startup, to test the feasibility, the directory contains test code to compute the norm of a vector of numbers by C code.

```
function normC ( n : integer32;         -- n is the dimension
                 x : C_Double_Array;    -- contains 2*n doubles
                 y : C_Double_Array )   -- on return is y(0)
               return integer32;
pragma import(C, normC, "cpu2norm_d_in_c");
```

The function normC can be used as an Ada function. The connection with C is defined by the pragma import where cpu2norm_d_in_c is the name of the file which contains the definition of the C code of the C function. The type C_Double_Array is defined in the State subdirectory of the CtoPHC directory.

4.2.9 Multitasking

The Ada tasking mechanisms allows to define shared memory parallel programs at a high level. Tasks in Ada are mapped to kernel threads. There are two main applications defined in the Tasking directory.

Given a queue of path tracking jobs, the tasks are arranged in a work crew model to execute all jobs. Dynamic load balancing is achieved as tasks, when done with their current job, grab the next job from the queue. Synchronization overhead is minimal, as only the movement of the current pointer in the job queue happens in a critical section. This parallel work crew path tracking scheme is implemented for regular homotopies and polyhedral homotopies.

Another application of multitasking is pipelining. Polyhedral homotopies start at initial form systems computed by the mixed cells. For large polynomial systems, the computation of the mixed volume could be a bottleneck for the parallel

execution. A pipelined multitasked implementation of the polyhedral homotopies combines the tracking of all paths with the mixed cell computation as follows. One task computes the mixed cells and appends the mixed cells to the job queue. Other tasks take the mixed cells as the jobs to solve the random coefficient system. As soon as one mixed cells is available in the queue, the path tracking can start.

4.2.10 The Main Program

The directory `Main` contains the main program, called `dispatch` because its main function is to dispatch the options given at the command line to the specific procedures.

The code for the blackbox solver (invoked by `phc -b`) is defined by the packages `black_box_solvers` and `black_box_root_counters`.

A very specific solver is defined by the file `use_phc.adb`, mainly as an example how the code could be customized for one particular application. The code is below:

```
with text_io;                              use text_io;
with Standard_Natural_Numbers;             use Standard_Natural_Numbers;
with Standard_Complex_Poly_Systems;        use Standard_Complex_Poly_Systems;
with Standard_Complex_Poly_Systems_io;     use Standard_Complex_Poly_Systems_io;
with Standard_Complex_Solutions;           use Standard_Complex_Solutions;
with PHCpack;

procedure use_phc is

  infile,outfile : file_type;         -- input and output file
  p,q : Link_to_Poly_Sys;             -- target and start system
  mixed_volume : natural32;           -- root count is mixed volume
  sols : Solution_List;               -- list of solutions

begin
  Open(infile,in_file,"test.in");
  get(infile,p);
  Create(outfile,out_file,"test.out");
  put(outfile,p.all);
  q := new Poly_Sys(p'range);
  PHCpack.Static_Lifting(outfile,p.all,mixed_volume,q.all,sols);
  PHCpack.Artificial_Parameter_Continuation(outfile,p.all,q.all,sols);
  PHCpack.Refine_Roots(outfile,p.all,sols);
end use_phc;
```

4.3 Numbers, Linear Algebra, Polynomials and Polytopes

In this section we take a closer look at the `Math_Lib` directory, which defines the basic mathematical data structures and operations.

4.3.1 Numbers

The machine numbers are divided in two categories: integer and float. For the integer types, we distinguish between the 32-bit and 64-bit versions, between natural and integer numbers. The following types are defined: `natural32`, `natural64`, `integer32`, and `integer64`. For the float types, we have single precision and double precision, defined respectively as `single_float` and `double_float`. The renaming of the hardware number types ensures the independence of pre-defined number types.

For polynomial system solving, our default field is the field of complex numbers. The real and imaginary part of a complex number are floating-point coefficients. The homotopy algorithms depend on the choice of random constants. Random number generators are defined. The default seed for the random number generators is the process identification number. For reproducible runs, the user can set the seed to a fixed number.

Multiprecision numbers are implemented as arrays of machine integers. Elementary school algorithms defined the arithmetic. The implementation of the floating-point multiprecision numbers is directly based on the multiprecision integer numbers, for the fraction and the exponent part of the multiprecision float. The precision of each multiprecision number can be adjusted when needed, which is an advantage. Mixed-precision arithmetical operations are supported. The disadvantage imposed by this flexibility is the frequent memory allocation and deallocation, which makes this type of arbitrary multiprecision arithmetic unsuitable for shared memory parallelism.

The directory `Numbers` contains definitions of abstract rings, domains, and fields. These abstract classes are useful to define composite generic types. Multiprecision complex numbers are defined via the instantiation of a generic complex numbers package.

4.3.2 Quad Doubles

The directory `QD` provides the double double and quad double arithmetic, based on the QDlib package of Y. Hida, X. S. Li, and D. H. Bailey.

Compared to arbitrary multiprecision arithmetic, double double and quad double numbers exploit the floating-point hardware and have a simple memory management. While arbitrary multiprecision numbers are allocated via the heap, the two doubles of a double double and the four doubles of a quad double use the stack. Thus the QD library is very well suited for shared memory parallelism. Another advantage is the predictable cost overhead. Working with double doubles has a similar cost overhead as working with complex numbers. Computations with double doubles are about five to eight times slower compared to computations in double precision. With quad doubles, computations that took seconds in double precision can turn into minutes.

The code in QDlib was hand translated into Ada. The directory contains the original C versions for comparison and verification of correctness.

4.3.3 Vectors and Matrices

The directories `Vectors` and `Matrices` contain the definitions of respectively all vector and matrix types. In both directories, generic packages are defined, which allow to specify the ring of numbers (natural32, integer32, natural64, integer64) or the number fields (double, double double, quad double, or arbitrary multiprecision). Input and output for all types is provided.

Although both `Vectors` and `Matrices` are basic data structures, random number generators are provided, to generate vectors and matrices of random numbers. The test procedures check the basic arithmetical operations.

The directory `Vectors` defines vectors of vectors and vectors of matrices are defined in the directory `Matrices`.

4.3.4 Linear Systems with Integer Coefficients

The problem considered in the directory `Divisors` is the manipulation of matrices with integer coefficients.

With the greatest common divisor we can define unimodular coordinate transformations to compute an upper triangular form of a matrix with integer coefficients. Such form is call the Hermite normal form. The diagonalization process results in the Smith normal form.

Even if the input matrices have small integer coefficients, the size of the integers in the unimodular coordinate transformations can outgrow the size of the hardware integers. Therefore, multiprecision versions of the normal forms are provided.

This integer linear algebra is applied in the computation of the volumes of the mixed cells of subdivisions of Newton polytopes.

4.3.5 Linear Systems with Floating-Point Coefficients

The directory `Reduction` contains several matrix factorizations as common in numerical linear algebra.

The LU factorization is based on the `lufac`, `lufco`, and `lusolve` of the F77 LINPACK libary. The Fortran77 code was translated into Ada and extended with versions for double double, quad double, and arbitrary multiprecision; both for real and complex number types.

To solve overdetermined linear systems in the least squares sense, packages are provided for the QR decomposition. Also the Singular Value Decomposition (SVD) is implemented, for all precisions, and for real and complex number types.

To implement a variable precision Newton's method, there are variable precision linear system solvers. Given the desired accuracy, the variable precision linear system solver sets the working precision based on a condition number estimate.

4.3.6 Polynomials in Several Variables

Multivariable polynomials and polynomial systems are defined in the directory `Polynomials`. In addition to ordinary polynomials, polynomials with integer exponents, so-called Laurent polynomials, are defined as well. In solving Laurent polynomials, solutions with zero coordinates are excluded.

There are packages to read and parse polynomials in symbolic form, from the standard input, from a file, and from a string. Also the writing of polynomials works for standard output, to file, or to string. The parsing from strings is especially important in connection with the use of multiprecision arithmetic. An innocently looking constant such as `0.1` has no exact binary representation and will have a nonzero representation error, dependent on the working precision with which it was evaluated. The input system given by the user is stored in its string representation. When later in the program, the user wants to increase the working precision, all mathematical constants are evaluated anew in the higher working precision. Numerical algorithms solve nearby problems not exact ones. Increasing the working precision may increase only the distance to the exact input problem.

The symbolic form of a polynomial system makes the program user friendly. For some applications, a flat representation of a polynomial into a tuple of coefficients and exponents is a more convenient data structure, both for internal and external use, for a more direct interface. In addition to the symbolic format, code is available to represent a polynomial system in a tableau format. For example,

```
2
3
 1.00000000000000E+00 0.00000000000000E+00 2 0
 4.00000000000000E+00 0.00000000000000E+00 0 2
-4.00000000000000E+00 0.00000000000000E+00 0 0
2
 2.00000000000000E+00 0.00000000000000E+00 0 2
-1.00000000000000E+00 0.00000000000000E+00 1 0
```

is the tableau format of the system, in symbolic format:

```
2
 x**2 + 4*y**2 - 4;
        2*y**2 - x;
```

where the variables are represented by the symbols `x` and `y`. In the tableau format, the term `4*y**2` is represented by

```
4.00000000000000E+00 0.00000000000000E+00 0 2
```

where the coefficient appears first as a complex number, as a sequence of two doubles, its real and imaginary part. The monomial `y**2` is represented as `0 2` as the `y` is the second variable which appeared in the symbolic format of the system and 2 is its exponent.

4.3.7 Nested Horner Forms for Evaluation

Because the evaluation and differentiation of polynomials can be just as expensive as solving a linear system in the application of Newton's method, the distributed list of terms in a polynomial is converted into a nested Horner form, for efficient evaluation. The directory `Functions` provides specific data structures to construct and evaluate the nested Horner forms.

For polynomial systems of low degrees and dimensions, the change in data structure from a linked list of terms into a recursive array structure yields significant improvements on the memory access, in addition to the saved multiplications. For larger polynomial systems, methods of algorithmic differentiation are required, as provided in the directory `Circuits`.

4.3.8 Support Sets and Linear Programming

Given a list of vectors with integer coefficients, via linear programming we can extract from the list those points which are vertex points of the polytope spanned by the points in the list. Another application of linear programming is the computation of all k-dimensional faces of the polytope. The directory `Supports` provides the primitive operations for the volume computations in the polyhedral root counts.

4.3.9 Circuits for Algorithmic Differentiation

The directory `Circuits` contains implementations of the algorithms which evaluate and differentiate polynomials in several variables using the reverse mode of algorithmic differentiation.

The current state of the code in this directory is still experimental, mostly geared towards algorithmic correctness rather than performance. An efficient implementation is available in the GPU part of the source code.

4.3.10 Truncated Power Series

Similar to Taylor series approximations for general functions, we can approximate roots of polynomials in a parameter by series. The directory `Series` defines truncated power series with complex numbers as coefficients. Composite types are vectors, matrices, and polynomials where the coefficients are series.

The division of two truncated power series is computed via the solution of a triangular linear system. So we can have a field and we can solve linear systems over this field of truncated power series. However to work efficiently, instead of working with vectors and matrices of power series, we apply linearization and consider series where the coefficients are vectors and matrices.

The directory exports packages to solve linear systems where the coefficient matrix is a power series of matrix coefficients. We can solve such linear systems with LU factorization, or for overdetermined problems we solve in the least squares sense, either with a QR or an SVD decomposition. To solve Hermite-Laurent interpolation problems, a lower triangular echelon form is provided.

4.4 Homotopies, Newton's Method, and Path Trackers

The directory `Deformations` provides data structures for solutions and polynomial homotopies. Newton's method serves as a corrector in the path trackers and has been modified by deflation to compute isolated singularities. Predictors are defined in the `Curves` subdirectory and polyhedral end games are provided in the subdirectory `End_Games`. Path trackers for solutions defined by artificial-parameter homotopies and natural-parameters are provided respectively in the subdirectories `Trackers` and `Sweep`.

4.4.1 Solutions of Systems and Homotopies

The second most important data structures, after the polynomials, are the data structures to represent solutions of polynomial systems. There are three parts in the library.

1. The data structure for solutions are defined for double, double double, quad double, and general arbitrary multiprecision. The reading and writing of the solutions makes use of the symbol table, so the coordinates of the solutions are connected to the symbols used to represent the variables in the system. The input and output is implemented for the standard input and output, for files, and for strings.
2. The directory contains functions to filter solutions subject to certain given criteria. For example, one such criterion is whether the solution is real or not. To process huge lists of solutions, in particular to check whether all solutions are distinct from each other, a double hash function on a solution list fills a quad tree.
3. To export solutions to other programs, format conversions are implemented, in particular for Maple and Python. For the computer algebra system Maple, a solution is represented as a list of equations. For the scripting language Python, a solution is formatted into Python's dictionary data structure.

Conversions between solutions in various levels of precision are available for the variable precision Newton's method.

4.4.2 Polynomial Homotopies

The `Homotopy` directory provides packages to define polynomial homotopies in double, double double, quad double, and arbitrary multiprecision. These homotopy packages encapsulate the efficient evaluation data structures.

Stable mixed volumes allow to count the solutions with zero coordinates separately from the other solutions. For the separate computation of the solutions with zero coordinates, as defined by the zero type of the stable mixed cells, special, so-called stable homotopies are implemented. In these homotopies, the variables which correspond to zero coordinates are removed so solutions with zero coordinates are thus computed more efficiently than the solution with all their coordinates different from zero.

This directory also provides methods to scale the coefficients of polynomial systems via an optimization problem to recenter the magnitudes of the coefficients. Another preconditioner is the reduction of the degrees of the polynomial via linear row reduction and selective replacement with S-polynomials.

The blackbox solver recognizes linear systems as a particular case. Packages to check whether a given polynomial system is linear and then to call a linear solver are provided in this directory.

4.4.3 Newton's Method and Deflation for Isolated Singularities

The directory `Newton` has its focus on the implementation of Newton's method and the modification to locate isolated singularities accurately with deflation.

Newton's method is applied as the corrector in the path trackers and to verify and refine solutions at the end of the path tracking. The method is available in double, double double, quad double, and arbitrary multiprecision. The variable precision Newton's method estimates the condition number of the polynomial evaluation problem and the condition

number of the Jacobian matrix, both at the current approximation of the solution, to set the precision in order to guarantee the desired number of correct decimal places in the answer.

To restore the quadratic convergence of Newton's method in case the Jacobian matrix is no longer of full rank, the deflation operator appends random combinations of the derivatives recursively, until the extended Jacobian matrix becomes of full rank. The rank is computed using the singular value decomposition. Derivatives are computed in an efficient hierarchy encoded in a tree data structure.

4.4.4 Curves, Univariate Solvers, and Extrapolators

The directory `Curves` contains an implementation of the method of Weierstrass (also called the Durand-Kerner method) to compute all roots of a polynomial in one variable. A polynomial in one variable is another special case of the blackbox system solver.

Divided differences are computed to extrapolate the solutions for the predictors. The higher order extrapolating predictors are available in double, double double, quad double, and arbitrary multiprecision. Univariate polynomial solvers are used to sample plane algebraic curves and to test the higher-order extrapolators.

The directory provides packages to run Newton's method to compute series solutions of polynomial homotopies, both in the basic version with operator overloading and the more efficient version with linearization.

4.4.5 Polyhedral End Games

Deciding whether a solution path diverges to infinity is a critical decision. Solutions with coordinates of large magnitude are difficult to distinguish from solutions at infinity.

The directory `End_Games` contains code for a polyhedral end game, implementing Bernshtein second theorem: if there are fewer solutions than the mixed volume, then there are solutions of initial form systems, supported on faces of the Newton polynomials of the given system.

In a polyhedral end game, the direction of the diverging path gives the inner normal which defines the initial form system that has a solution with all its coordinates different from zero. What complicates the computation of this inner normal is the presence of winding numbers larger than one. If the step size is decreased in a geometric rate, then the winding number can be computed with extrapolation. The certificate for a diverging path consists of the inner normal which defines an initial form system where every equation has at least two monomials with a nonzero coefficient. In addition, the end point of the diverging path is (after a proper unimodular coordinate transformation) a solution of the initial form system.

The polyhedral end games are implemented in double, double double, and quad double precision.

4.4.6 Path Trackers for Artificial-Parameter Homotopies

In an artificial-parameter homotopy, singular solutions can only occur at the end of the solution paths. There are two different parts in the directory `Trackers`, corresponding to the different ways to run a path tracker, depending on the level of control.

In the first, most conventional way of running a path tracker, the procedure which implements the path tracker gets called with data and various execution parameters. Then the procedure takes control of the execution thread and control is only returned when the end of the solution path has been reached. This first way is available in double, double double, and quad double precsion. The application of the QR decomposition in the corrector leads to the capability of tracking paths defined by overdetermined polynomial homotopies.

In the second way of running a path tracker, the path tracker is initialized with a start solution and some initial settings of the execution parameters. The procedure that calls the path tracker wants only the next point on the path and the path tracker is then restarted when another next point is needed. This type of path tracker is particularly useful in a

scripting environment when the user wants to visualize the results of the path tracker and the responsibility for the memory management of all data along a solution path is the responsibility of the calling procedure, not of the path tracker.

A preliminary prototype of a variable precision path tracker has been implemented. Depending on the condition numbers of the evaluation and the Jacobian matrix, the precision is adjusted to ensure a desired number of correct decimal places.

4.4.7 Sweeping for Singularities

In a natural parameter homotopy, singular points along the solution paths are expected to occur. A path tracker for a natural parameter homotopy has two tasks: the detection and the accurate location of singular solutions. The directory `Sweep` provides packages to compute accurately quadratic turning points and to search for general singularities along a solution path, in double, double double, and quad double precision.

If one is only interested in the real solutions, then tracking the solution paths in real instead of complex arithmetic can go about five times faster. One has to tracker fewer paths, as the paths with nonzero imaginary coordinates appear in pairs, thus it suffices to track only one path in the complex conjugated pair. For sufficiently generic real coefficients, the only type of singular solutions that may occur are quadratic turning points. A quadratic turning point is where a real path turns back in the direction of an increasing continuation parameter. At a quadratic turning point, the real path touches the complex conjugated pair of paths where their imaginary parts become zero. If one forces the continuation parameter to increase, then the real path turns complex or vice versa, a complex path turns real. Quadratic turning points can be computed efficiently via an arc-length parameter continuation and the application of a shooting method when the orientation of the tangent vector flips.

The detection and accurate location of general types of singular solutions is much more difficult. If the sign of the determinant of the Jacobian matrix flips, then we passed a singularity. But the determinant of the Jacobian matrix may remain of the same sign before and after passing through a singular solution. The criterion implemented monitors the concavity of the determinant of the Jacobian matrix. If the value of the determinant increases in magnitude after a decrease, then we may have missed a singular solution and we turn back with a finer granularity, in an attempt to locate the singularity.

4.4.8 Polynomial Continuation

The directory `Continuation` provides data structure and data management procedures to organize the application of path trackers to the solution paths defined by a polynomial homotopy.

The interactive tuning of the settings and tolerances for the path trackers are defined in this folder. Several different levels of the amount of output information during the path trackers are possible, going from nothing to all data.

4.5 Root Counts and Start Systems

An important feature of the code is the automatic construction of a good start system in an artificial-parameter homotopy. For a start system to be good, it needs to resemble as much as possible the structure of the target system.

For generic polynomial systems, where the coefficients are sufficiently generic, the mixed volume of the Newton polytopes offers an exact count on the number of isolated solutions, where all coordinates are nonzero.

4.5.1 Linear-Product Start Systems

The directory `Product` contains packages to construct start systems based on the degree structure of a polynomial system. There are two main categories of start systems.

1. Total degree start systems. The classical theorem of Bezout states that the product of the degrees of the polynomials in the system gives an upper bound on the number of isolated solutions. A total degree start system consists of a decoupled system, where the *k*-th polynomial equation in the start system equals $x_k^{d_k} - c_k = 0$, where d_k is the degree of the *k*-th polynomial in the target system and where c_k is some random nonzero complex coefficient.
2. Linear-product start systems. Every polynomial in a linear-product start system is a product of linear polynomials with random coefficients. Which variables appear with a nonzero coefficient in the linear polynomials is determined in three ways. The first way is one single partition of the set of unknowns. In the second way, a different partition may be used for each different polynomial in the system. For general linear-product start systems, the structure of each polynomial is represented by a sequence of sets of variables. Every variable should appear in as many sets in the sequence as its degree in the polynomial.

Lexicographic enumeration of the solutions of a start system is supported. By this enumeration, it is not necessary to compute the entire solution set of a start system in memory, as one can ask for the computation of a particular start solution.

The generalized Bezout bounds are a special case of the polyhedral root counts. In case the Newton polytopes can be written as the sum of simplices, the generalized Bezout bound matches the mixed volume.

4.5.2 Binomials are Polynomials with Two Terms

The sparsest (Laurent) polynomial systems which allow solutions with all coordinates different from zero are systems where the polynomials have exactly two monomials with a nonzero coefficient. We call such polynomials binomials and systems of binomials are binomial systems. The computation of all solutions with nonzero coordinates happens via a unimodular coordinate transformation. An extension of a binomial system is a simplicial system: the support of a simplicial system is a simplex. The directory `Binomials` provides solvers for binomial and simplicial systems.

Binomial and simplicial systems are start systems in a polyhedral homotopy, induced by a generic lifting, where all mixed cells in the regular subdivision are fine. A simplicial system is reduced to a binomial system via a diagonalization of its coefficient matrix. Binomial systems are solved via a Hermite normal form on the matrix of exponent vectors. Because the solution of binomial and simplicial systems does not involve any path tracking (just linear algebra), the systems can be solved much faster and the blackbox solver treats such systems as a special case.

Even though as the exponents in the binomial systems might be small in size, the size of the coefficients in the unimodular coordinate transformations may result in relatively high exponents. This height of the exponents could lead to overflow in the floating-point exponentiation of the partial results in the forward substitution. Therefore, for a numerically stable solution of a binomial system, we separate the radii from the arguments in the right hand side constant coefficients. This scaled solving prevents overflow.

Underdetermined binomial systems are rational: their positive dimensional solution set admits an explicit parameter representation. Packages are defined to represent and manipulate monomial maps. Monomial maps define the leading terms of a Puiseux series expansion of a positive dimensional solution set.

4.5.3 Implicit Lifting

The directory `Implift` contains the code for the original version of the polyhedral homotopies, as provided in the constructive proof of D. N. Bernshtein's paper. The polyhedral homotopies induced by an implicit lifting are based on the following formula to compute the mixed volume of the Newton polytopes. Given a tuple of Newton polytopes $\mathbf{P} = (P_1, P_2, \ldots, P_n)$, the mixed volume $V_n(\mathbf{P})$ can be computed via the formula

$$V_n(P_1, P_2, \ldots, P_n) = \sum_{\substack{\mathbf{v} \in \mathbb{Z}^n \\ \gcd(\mathbf{v}) = 1}} p_1(\mathbf{v})\, V_{n-1}(\partial_{\mathbf{v}} P_2, \ldots, \partial_{\mathbf{v}} P_n),$$

where p_1 is the support function for P_1 and V_1 is the length of a line segment. Vectors $\mathbf{v}$ are normalized so the components of $\mathbf{v}$ have their greatest common divisor equal to one.

Functionality is provided to extract the vertex points from the support sets of the polynomials in the system. Polyhedral homotopies may be combined with linear-product start systems: for some polynomials we use a linear-product structure and for the remaining polynomials a random coefficient start system is solved.

4.5.4 Static Lifting

The static lifting as implemented in the code in the directory `Stalift` is so named in contrast with dynamic lifting. Static lifting applies before the mixed volume computation. Both integer valued and floating-point valued lifting functions are supported.

One particular lifting leads to the computation of the stable mixed volume. While the mixed volume often excludes solutions with zero coordinates, the stable mixed volume is an upper bound for all isolated solutions, also for solutions with zero coordinates.

4.5.5 Dynamic Lifting

Volumes are monotone increasing in the size of the polytopes: the more vertices in a polytope, the larger the volume. One way to build a triangulation of a polytopes is by placing the points one after the other. The next point can be lifted sufficiently high so that the existing simplices in the triangulation remain invariant. Applied in connection with a polyhedral homotopy, one can solve polynomial systems monomial by monomial.

Dynamic lifting is applied to compute a triangulation of the Cayley embedding, which leads to the Minkowski polynomial. Given a tuple of polytopes $(P_1, P_2, \ldots, P_n)$, Minkowski showed that the volume of the linear combination $\lambda_1 P_1 + \lambda_2 P_2 + \cdots + \lambda_n P_n$ is a homogeneous polynomial of degree n in the variables λ_1, λ_2, and λ_n. The coefficients of this homogeneous polynomial are mixed volumes of the polytopes in the tuple.

4.5.6 Exploitation of Permutation Symmetry

In a polynomial homotopy where every system, for every value of the parameter, has the same permutation symmetry, it suffices to track only the generating solution paths. The directory `Symmetry` provides support to construct symmetric start systems, given the generators of the permutation group.

4.5.7 MixedVol to Compute Mixed Volumes Fast

The directory `MixedVol` contains an Ada translation of the MixedVol algorithm, archived by ACM TOMS as Algorithm 846, developed by Tangan Gao, T. Y. Li and Mengnien Wu.

The C version of the code (written by Yan Zhuang) is contained for comparison and correctness verification.

The code is restricted for randomly generated lifting values.

4.5.8 The Newton-Puiseux Method

The directory `Puiseux` contains an implementation of the Newton-Puiseux method to compute power series expansions for all solution curves of a regular polynomial system. In this context, a polynomial system is regular if its coefficients are sufficiently generic, so its initial form systems have no singular solutions.

The code in this directory applies the integer lifting applied to compute the mixed volume of a tuple of Newton polytopes. The key is to use as values of the lifting the powers of the variable of the parameter in the series. Newton's method on power series provides the series expansion for the solution curves.

4.6 Determinantal Systems and Schubert Problems

A Schubert problem gives rise to a so-called determinantal system, a system where the polynomials are obtained via minor expansions of a matrix. That matrix then represents the intersection condition of a given plane with an unknown plane. In a general Schubert problem we require that a *k*-dimensional plane intersects a sequence of spaces nontrivially in particular dimensions.

The directory `Schubert` consists in three parts, described briefly in the sections below.

4.6.1 SAGBI Homotopies to Solve Pieri Problems

SAGBI stands for Subalgebra Analogue to Groebner Basis for Ideals. The directory `SAGBI` provides packages to define SAGBI homotopies to compute all *k*-planes which meet as many as $m \times p$ general *m*-planes in a space of dimension $m + p$. The SAGBI homotopies were applied to investigate a conjecture concerning particular input *m*-planes for which all solution *k*-planes are real.

Packages are available to manipulate brackets. Brackets represent intersection conditions and encode selection of columns in minor expansions. A particular application is the symbolic encoding of the Laplace expansion to compute the determinant of a matrix. The straightening law for brackets leads to a Groebner basis for the Grassmannian. This Groebner basis defines a flat deformation which defines the SAGBI homotopy. The start system in the SAGBI homotopy is solved by a polynomial homotopy.

4.6.2 Pieri Homotopies

The directory `Pieri` offers a more generic solution to solve Pieri problems. Pieri homotopies are capable to solve more general Pieri problems. For all these Pieri problems, there is a combinatorial root count which quickly gives the number of solutions to a generic Pieri problem.

4.6.3 Littlewood-Richardson Homotopies

General Schubert problems can be solved by a geometric Littlewood-Richardson rule, as implemented by the code in the directory `Induction`.

A general Schubert problem is given by a sequence of flags and a sequence of intersection conditions that must be satisfied by the *k*-plane solutions of the Schubert problem. The geometric Littlewood-Richardson rule to count the number of solutions is implemented by a checker board game. The stages in the game correspond to specific moves of the solutions with respect to the moving flag.

4.7 Positive Dimensional Solution Sets

This section describes the specific code to compute a numerical irreducible decomposition of a polynomial system. The directory `Components` have six subdirectors, which are briefly described in the next sections.

4.7.1 Witness Sets, Extrinsic and Intrinsic Trackers

The subdirectory `Samplers` contains the definition of the data structures to represent positive dimensional solution sets, the so-called witness set. A witness set contains the polynomial equations, as many random linear equations as the dimension of the set, and as many generic points (which satisfy the original polynomial equations and the random linear equations) as the degree of the solution set.

The extrinsic way to represent a witness set is formulated in the given equations, in the given variables. For a high dimensional solution set, the number of equations and variables almost doubles. For example, for a hypersurface, a solution set of dimension $n-1$, the extrinsic representation requires $2n-1$ equations and variables. This doubling of the dimension leads to an overhead of a factor of eight on the linear algebra operations when computing new points on the positive solution set.

The intrinsic way to represent a witness set computes a basis for the linear space spanned by the random linear equations. This basis consists of an offset point and as many directions as the dimension of the linear space. Then the number of intrisic variables equals the dimension of the linear space. For a random line to intersect a hyperface, the intrisic representation reduces to one variable and computing new generic points on a hypersurface is reduced to computing new solutions of a polynomial equation in one variable.

Unfortunately, the use of intrinsic coordinates, while reducing the number of equations and variables, increases the condition numbers of the witness points. To remedy the numerical conditioning of the intrinsic representation, tools to work with local coordinates are implemented. In local intrinsic coordinates, the offset point is the origin.

4.7.2 Equations for Solution Components

Once we have enough generic points on the positive dimensional solution components, we can compute equations for the components with the application of interpolation. Code for the interpolation is provided in the subdirectory `Interpolators`.

Three approaches have been implemented. The first direct approach solves a linear system, either with row reduction or in the least squares sense. The second technique applies a recursive bootstrapping method with generalized divided differences. Thirdly, the trace form leads to Newton interpolation.

Another application of interpolation is the computation of the linear span of a solution set. We know for instance that every quadratic space curve lies in a plane. With the linear equations that define this plane, an accurate representation for a quadratic space curve is obtained. With the linear span of a component, the cost to compute new generic points on a solution set is reduced.

4.7.3 Absolute Factorization into Irreducible Components

The problem considered in the `Factorization` directory takes a pure dimensional solution set on input, given as a witness set, and computes a witness set for every irreducible component. The *absolute* in the title of this section refers to the factorization over the complex numbers.

Three methods are implemented to decompose a pure dimensional solution set into irreducible components. The first method applies incremental interpolation at generic points, using polynomials of increasing degrees. Multiprecision becomes necessary when the degrees increase. The second method is more robust and can handle higher degree components without multiprecision. This method runs loops exploiting the monodromy, using the linear trace as the stop test. The third method enumerates all factorizations and prunes the enumeration tree with linear traces.

A particular case is the factorization of a multivariate polynomial, which is directly accessible from the blackbox solver.

4.7.4 Cascades of Homotopies and Diagonal Homotopies

The code in `Decomposition` aims to produce generic points on all pure dimensional components of the solution set of a polynomial system.

The first top down method applies cascades of homotopies, starting at the top dimensional solution set. With every added linear equation there is a slack variable. For solutions on the component intersected by the linear equations, all slack variables are zero. Solutions with zero slack variables are generic points on the positive dimensional solution set. Solutions with nonzero slack variables are regular and serve as start solutions in a homotopy to compute generic

points on the lower dimensional solution sets. Every step in the cascade removes one linear equation. At the end of the cascade we have computed all isolated solutions.

The result of running a cascade of homotopies is list of candidate generic points, as some of the paths may have ended to higher dimensional solution sets. To filter those points, a homotopy membership test starts at a witness set and moves to another set of linear equations that pass through the test point. If the test point is among the new generic points, then the test point belongs to the solution set represented by the witness set.

The second bottom up method applies diagonal homotopies. A diagonal homotopy takes on input two witness sets and produces on output generic points on all parts of the intersection of the solution sets represented by the two witness sets. Two versions of the diagonal homotopy are implemented, once in extrinsic coordinates, and once in intrinsic coordinates.

4.7.5 An Equation-by-Equation Solver

Diagonal homotopies can be applied to solve polynomial systems incrementally, adding one equation after the other, and updating the data for the solution sets. An equation-by-equation solver is implemented in the directory `Solver`.

4.7.6 Tropicalization of Witness Sets

The asymptotics of witness sets lead to tropical geometry and generalizations of polyhedral methods from isolated solutions to positive dimensional solution sets.

The code in the directory `Tropical` collects a preliminary standalone implementation of a method to compute the tropical prevariety for low dimensional problems.

4.8 Organization of the C and C++ code

C code can be called from within Ada, as is the case with the realization of the feedback laws in the output placement problem, as defined in the `Feedback` directory. A C (or C++) function may call Ada code, as was done in the message passing code in the `MPI` directory.

Via the options of the main executable `phc` the user navigates through menus and the data is stored on files. The C interface defines a state machine with persistent objects. As an example for the state machine metaphor, consider a vending machine for snacks. The user deposits coins, makes a selection, and then retrieves the snacks. The solution of a polynomial system via the C library happens in the same manner. The user enters the polynomials, either from file or via their string representations, selects some algorithms, and then retrieves the solutions, either from file, or in strings.

4.8.1 The Main Gateway Function

The directory `Lib` defines the C interface libraries. In analogy with the single main executable `phc`, there is only one interface function which serves at the main gateway exporting the Ada functionality to the C and C++ programmers.

The header files in the definitions of the prototypes of the library functions typically start with the following declarations:

```
#ifdef compilewgpp
extern "C" void adainit( void );
extern "C" int _ada_use_c2phc ( int task, int *a, int *b, double *c );
extern "C" void adafinal( void );
#else
extern void adainit( void );
```

```
extern int _ada_use_c2phc ( int task, int *a, int *b, double *c );
extern void adafinal( void );
#endif
```

The `adainit` and `adafinal` are generated by the binder of the gnu-ada compiler, see the section on Calling Ada from C. They are required when the main program is not written in Ada. Before the first call of the Ada code, `adainit` must be executed and `adafinal` is required after the last call, before termination of the program.

4.8.2 Persistent Objects

The C (or C++) can pass data via files or strings. The definition of the data structures for the polynomials and solution lists should not be duplicated in C (or C++). Unless an explicit deallocation job is performed, the objects remain in memory after a call to the Ada code.

The blackbox solver is exported by the C program `phc_solve`. The version which prompts the user for input and output files starts as follows:

```
int input_output_on_files ( int precision )
{
   int fail,rc,nbtasks;

   if(precision == 0)
   {
      fail = syscon_read_standard_system();
      printf("\nThe system in the container : \n");
      fail = syscon_write_standard_system();
      printf("\nGive the number of tasks : "); scanf("%d",&nbtasks);
      fail = solve_system(&rc,nbtasks);
      printf("\nThe root count : %d\n",rc);
      printf("\nThe solutions :\n");
      fail = solcon_write_standard_solutions();
   }
```

The `precision` equal to zero is the default standard double precision. Other precisions that are supported are double double and quad double precision. If the number of tasks in `nbtasks` is a positive integer, then the shared multicore version of the path trackers is executed. The code below illustrates the use of persistent objects: after the call to `solve_system`, the solutions remain in main memory even though only the value of the root count is returned in `rc`. The solutions are printed with the call to `solcon_write_standard_solutions()`.

4.9 Message Passing

The shared memory parallelism is based on the tasking mechanism defined by the Ada language and implemented by the gnu-ada compiler. This section describes the distributed memory parallelism with message passing, using the MPI library.

The tracking of all solution paths is a pleasingly parallel computation as the paths can be tracked independently from each other. Some paths are more difficult to track than others and may require more time, so dynamic load balancing in a manager/worker paradigm often gives close to optimal speedups. The setup suggested by Fig. 4.1 is one wherein the manager solves the start system and then distributes the start solutions to the worker nodes.

The setup in Fig. 4.1 leads to a top down control in which the manager dictates the actions of the workers. A more flexible setup is suggested in Fig. 4.2: start solutions are computed or retrieved when needed by the workers.

The advantage of the inverted control in Fig. 4.2 over the more conventional setup in Fig. 4.1 is the immediate availability of solutions of the target system. Moreover, the inverted control in Fig. 4.2 does not require to store all start

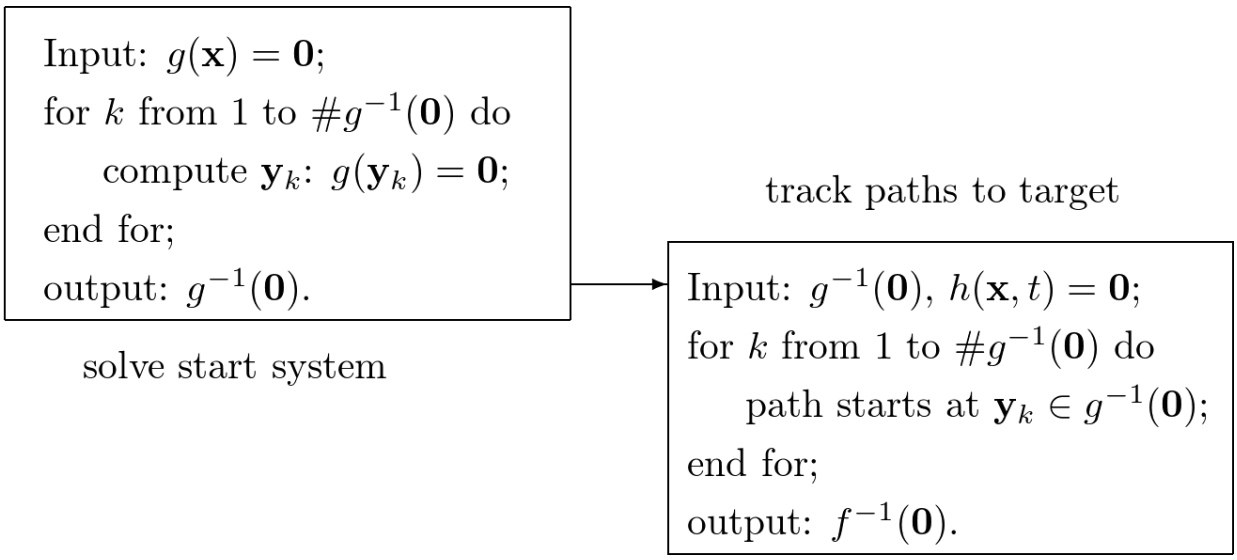

Fig. 4.1: A homotopy solver first solves the start system and then tracks all paths from start to target.

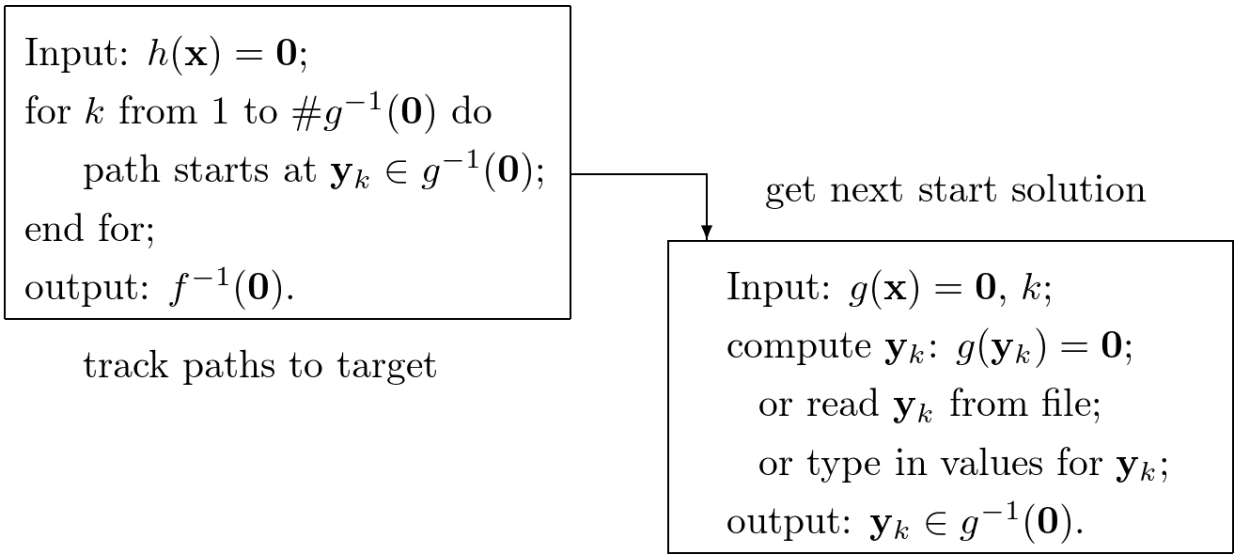

Fig. 4.2: The path tracker in a homotopy solver calls for the next solution of the start system.

solutions. For large polynomial systems, the number of start solutions may be too large to store in the main memory of one node.

4.10 GPU Acceleration

The acceleration with Graphics Processing Units (GPUs) is coded with the NVIDIA compiler. GPUs are designed for data parallel applications. Their execution model is single instruction multiple data: the same instruction is executed on many different data elements. Unlike shared memory parallelism with threads on multicore processors, to fully occupy a GPU, one must launch ten thousands of threads.

Polynomial homotopy continuation methods can take advantage of GPUs by the evaluation and differentiation of polynomials as required in the frequent application of Newton's method. The reverse mode of algorithmic differentiation applied to the monomials with appear with a nonzero coefficient in the polynomials provides sufficient parallelism and a granularity fine enough for the data parallel execution model. The same arithmetic circuits to evaluate and differentiate monomials are applied to different solutions when tracking many solution paths. For the tracking of one path in large enough dimension, different threads collaborate in the evaluation and differentiation algorithms.

To introduce the evaluation and differentiation algorithms consider Fig. 4.3 and Fig. 4.4 to compute the product of four variables and its gradient. Observe that results from the evaluation can be recycled in the computation of all partial derivatives.

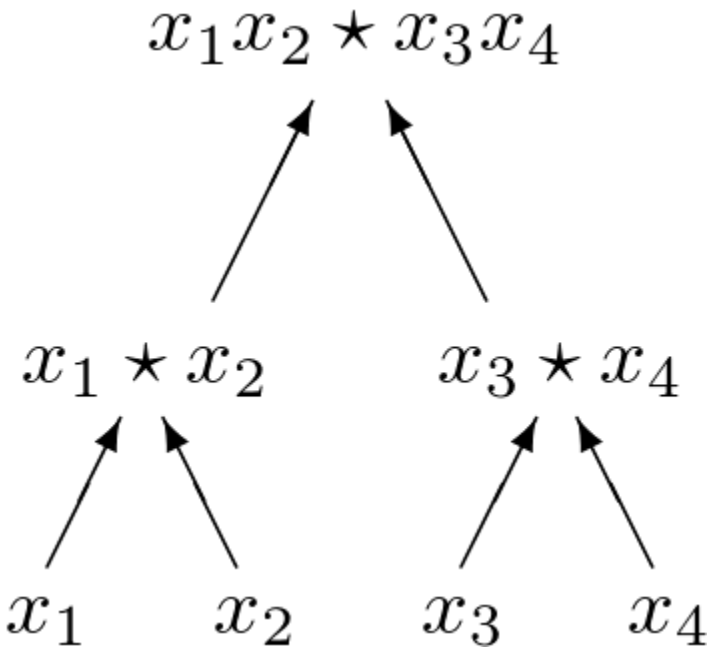

Fig. 4.3: An arithmetic circuit to evaluate the product of four variables x_1, x_2, x_3, and x_4.

The computation of the gradient of $x_1x_2 \cdots x_8$ is illustrated in Fig. 4.5.

4.11 The Web Interface

The directory `cgi` in the source code contains the Python CGI scripts to define a basic web interface.

The interface is entirely constructed in Python, the `index.html` directs the user to the `login.py` script in the `cgi-bin` directory. Images, the logo, and demonstration screenshots are contained in the `imag` directory. The

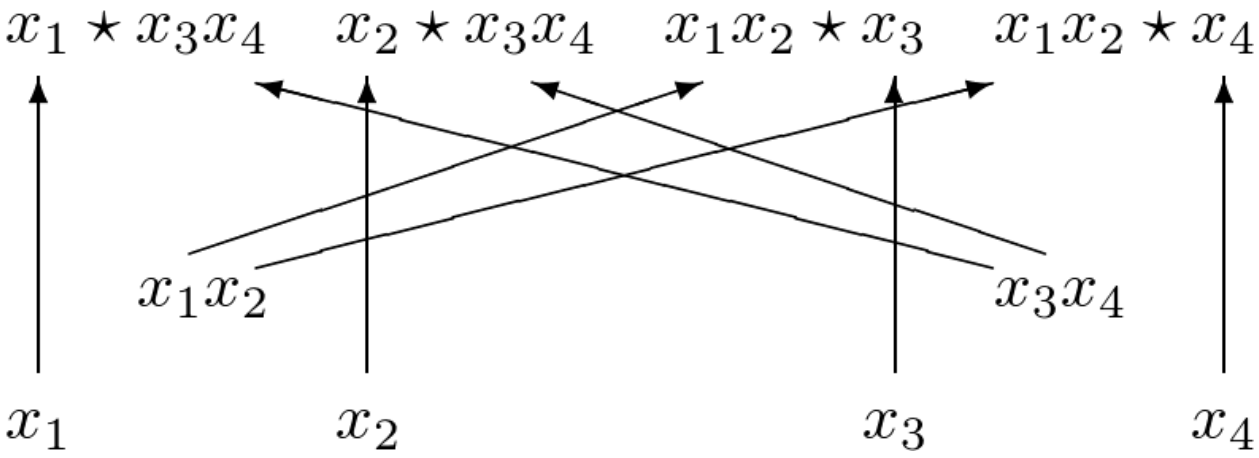

Fig. 4.4: An arithmetic circuit to compute the gradient of the product $x_1x_2x_3x_4$.

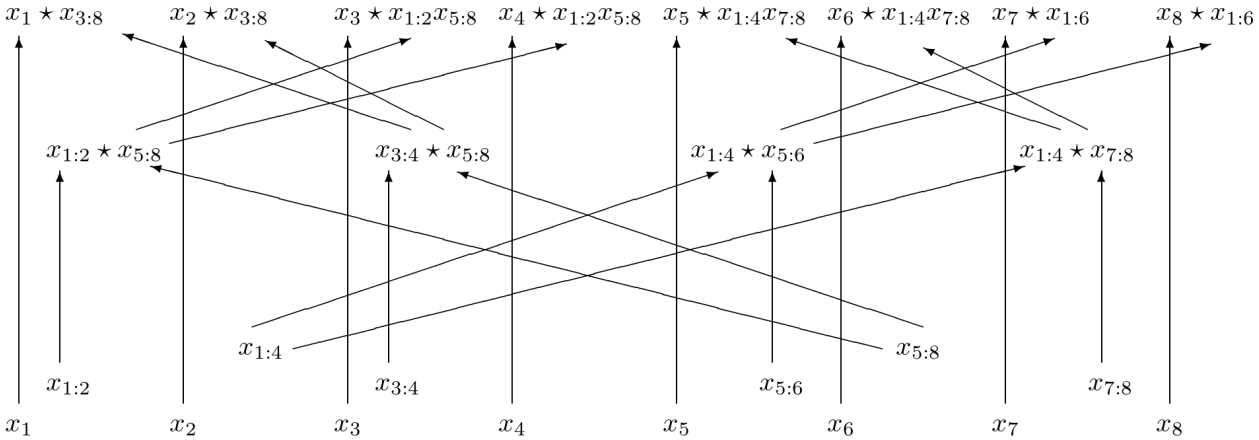

Fig. 4.5: An arithmetic circuit to compute the gradient of the product of eight variables x_1, x_2, ..., and x_8.

directory `style` collects the style files. Data from users is in the directory `users`.

4.12 The Python Package phcpy

The package phcpy provides a scripting interface. For its functionality phcpy depends mainly on the C interface and that was done on purpose: as the Python package grows, so does the C interface.

There are several other scripting interfaces to PHCpack: to the computer algebra system Maple (PHCmaple), PHClab for MATLAB and Octave, and for Macaulay2: PHCpack.m2. These other interfaces rely only on the executable version of the program.

Another major difference between phcpy and other scripting interface is the scope of exported functionality. The main goal of phcpy is to export all functionality of `phc` to the Python programmer. The development of phcpy can be viewed as a modernization of the PHCpack code, bringing it into Python's growing computational ecosystem.

The scripting interface to PHCpack has its own documentation.

Index

Q

R

S

T

V

W